Master‘s Thesis

ORCHID

Entwicklung eines Toolkits zur Integration der Arduino Microcontroller-Plattform in Softwareprojekte

AUTOR

Stefan Matyba
Medieninformatik Master
Matrikelnummer 758356

BETREUUNG DURCH

Beuth Hochschule für Technik Berlin
Luxemburger Straße 10
13353 Berlin

Betreuerin: Prof. Dr. Karin Schiele

Gutachter: Prof. Dr.-Ing. René Görlich

Inhaltsverzeichnis

1 Einleitung 1
1.1 Gliederung 1
1.2 Dokumentkonventionen 2
1.3 Englische Begriffe und Namen 3
1.4 Namensgebung 3
1.5 Begriffsdefinitionen 3
2 Fachliches Umfeld 4
2.1 Grundlagen Microcontroller 4
2.1.1 Microcontroller allgemein 4
2.1.2 Die Arduino Microcontroller-Plattform 5
2.2 Grundlagen HTTP 9
2.2.1 Allgemeines 9
2.2.2 Anfragen (Requests) 10
2.2.3 HTTP-Header 12
2.2.4 HTTP Status-Codes 12
2.2.5 Verwendung von HTTP im ORCHID Toolkit 13
2.3 Technologische Grundlagen 14
2.3.1 Mono 14
2.3.2 Arduino 15
3 Lösungsansätze 17
3.1 Marktanalyse 17
3.1.1 Allgemeines 17
3.1.2 EZ Control 17
3.1.3 digitalSTROM.org 18
3.1.4 Pachube.com 19
3.1.5 Fazit 20
3.2 Angestrebtes Ergebnis 20
3.2.1 Designkriterien 20
3.2.2 Technologiewahl 21
4 Anforderungsdefinition 23
4.1 Zielbestimmung 23
4.1.1 Musskriterien 23
4.1.2 Wunschkriterien 25
4.1.3 Abgrenzungskriterien 25
4.2 Produkteinsatz 26

4.2.1 Anwendungsbereich ... 26
4.2.2 Zielgruppe ... 26
4.2.3 Lizenzbedingungen ... 26
4.2.4 Einsatzmöglichkeiten ... 27
4.3 Produktanforderungen ... 29
4.3.1 Softwareanforderungen ... 29
4.3.2 Hardwareanforderungen ... 29
4.3.3 Produktschnittstellen ... 30
4.4 Produktfunktionen ... 30
4.4.1 Funktionen der Server-Komponente ... 31
4.4.2 Funktionen der Client-Bibliothek ... 39
4.5 Produktdaten ... 43
4.5.1 Server-Komponente ... 43
4.5.2 Client-Bibliothek ... 45
4.6 Produktleistung ... 46
4.6.1 Server-Komponente ... 46
4.6.2 Client-Bibliothek ... 47
4.7 Qualitätsanforderungen ... 48
4.7.1 Technische Qualitätsanforderungen ... 48
4.7.2 Wirtschaftliche Qualitätsanforderungen ... 48
4.7.3 Software-Ergonomie ... 49
5 Systementwurf ... 50
5.1 Übersicht ... 50
5.1.1 Projektstruktur Server-Komponente ... 50
5.1.2 Projektstruktur Client-Bibliothek ... 52
5.2 Hierarchie in der Server-Komponente ... 53
5.3 Entwurfsmuster ... 54
5.3.1 Das Entwurfsmuster „Abstrakte Fabrik“ ... 54
5.3.2 Das Singleton-Entwurfsmuster ... 56
5.3.3 Dependency Injection ... 58
5.4 Verwendete Technologien ... 61
5.4.1 C# und das Mono Framework ... 61
5.4.2 Arduino und C/C++ ... 66
5.4.3 Arduino Microcontroller-Plattform ... 67
5.5 Frameworks und Tools ... 67
5.5.1 NLog ... 67

5.5.2 Ninject ... 70
5.5.3 Nini ... 72
5.5.4 Microsoft FxCop ... 75
5.5.5 Microsoft StyleCop ... 75
5.6 Klassen der Server-Komponente ... 77
5.6.1 Server und Daemons ... 77
5.6.2 Parser und Requests ... 78
5.6.3 Datenhaltungsklassen ... 78
5.6.4 Datenbankadapter ... 79
5.6.5 Dynamisches Typeloading ... 80
5.7 Klassen der Client-Bibliothek ... 81
5.8 Protokoll ... 81
5.8.1 Allgemeines ... 81
5.8.2 Mögliche Antworten eines Servers ... 82
5.8.3 Sensordaten hochladen ... 83
5.8.4 Heartbeat senden ... 84
5.8.5 Instruktion abrufen ... 84
5.8.6 Instruktion hochladen ... 87
5.8.7 Sensordaten abrufen ... 88
5.8.8 Client-Informationen abrufen ... 89
5.9 Abläufe und Verhaltensweisen ... 91
5.9.1 Multithreading ... 91
5.9.2 Behandlung von Anfragen in der Server-Komponente ... 93
5.10 Datenbank ... 101
6 Realisierung und Tests ... 106
6.1 Styleguide ... 106
6.1.1 Einrückungen und Klammerungen ... 106
6.1.2 Sprache und Namenskonventionen ... 107
6.2 Entwicklungsumgebung ... 107
6.3 Realisierungsdetails ... 108
6.3.1 Eindeutige Kennungen ... 108
6.3.2 Dynamisches Typeloading mit Ninject ... 109
6.3.3 Sicherheitsaspekte ... 112
6.4 Testfälle und Auswertung ... 114
6.4.1 Unit-Tests (Server-Komponente) ... 114
6.4.2 Unit-Tests (Client-Bibliothek) ... 115

6.4.3 Lasttests ... 115
6.5 Probleme während der Entwicklung ... 115
6.6 Offene Probleme ... 116
6.6.1 Sicherheitsbedenken ... 116
6.6.2 Stabilitätsbedenken ... 118
6.6.3 Stabilitätsbedenken (Ethernet-Bibliothek) ... 118
6.6.4 Speicherprobleme ... 118
7 Handbuch Server-Komponente ... 119
7.1 Systemvoraussetzungen ... 119
7.2 Installation ... 119
7.2.1 Installation der MySQL-Datenbank und des MySQL-Servers ... 120
7.3 Referenzierung ... 121
7.4 Minimalprojekt ... 122
7.5 Konfiguration ... 124
7.5.1 Kategorie „Dependencies“ ... 124
7.5.2 Kategorie „Server“ ... 127
7.5.3 Kategorie „Restrictions“ ... 127
7.5.4 Kategorie „Logging“ ... 128
7.5.5 Kategorie „Database“ ... 128
7.5.6 Kategorie „Locale“ ... 128
7.5.7 Kategorie „Verbosity“ ... 129
7.5.8 Kategorie „Debug“ ... 129
7.6 Aktivierung des Logging-Systems ... 129
7.7 Veränderung und Erweiterung der Server-Komponente ... 132
7.7.1 Veränderung ... 132
7.7.2 Erweiterung ... 135
8 Handbuch Client-Bibliothek ... 137
8.1 Systemvoraussetzungen ... 137
8.2 Installation ... 137
8.2.1 Installation der Arduino IDE ... 137
8.2.2 Installation der Client-Bibliothek ... 138
8.3 Referenzierung ... 139
8.4 Codebeispiele ... 139
8.4.1 Initialisierung ... 139
8.4.2 Senden von Heartbeats ... 142
8.4.3 Hochladen von Sensordaten ... 142

8.4.4 Abrufen von Instruktionen ... 143
8.4.5 Direktes Ausführen einer Instruktion ... 146
8.4.6 Rückgabewerte und Fehlercodes verwenden ... 146
8.5 Problembehebung ... 147
8.6 Veränderung und Erweiterung der Client-Bibliothek ... 147
8.6.1 Veränderung ... 148
8.6.2 Erweiterung ... 149
9 Handbuch Zusatztools ... 151
9.1 ggen ... 151
9.2 dkgen ... 151
10 Endbenutzerhandbuch ... 152
10.1 Einleitung ... 152
10.2 Einrichtung der virtuellen Maschine ... 152
10.3 Übersicht ... 153
10.4 Einrichtung der MySQL-Datenbank ... 154
10.5 Hinzufügen eines ORCHID Clients ... 155
10.5.1 Hinzufügen mit phpMyAdmin ... 156
10.5.2 Hinzufügen über die Kommandozeile ... 157
10.6 Konfiguration und Start des ORCHID Servers ... 157
10.7 Einbindung eines ORCHID Clients ... 158
10.8 Drittsoftware einbinden ... 162
10.8.1 Sensordaten abrufen ... 162
10.8.2 Instruktionen übermitteln ... 164
10.8.3 Client-Informationen abrufen ... 164
10.9 Zusammenfassung ... 165
11 Zusammenfassung ... 166
11.1 Rückblick ... 166
11.2 Ausblick ... 167
11.2.1 Nachrichten ... 167
11.2.2 Drittsoftware ... 167
11.2.3 Smartphone-Applikationen ... 167
11.2.4 Datenbank-Systeme ... 167
12 Anhang I - Literaturverzeichnis ... I
13 Anhang II – Abbildungsverzeichnis ... I
14 Anhang III – Tabellenverzeichnis ... I
15 Anhang IV – Listingverzeichnis ... I

16 Anhang V – Glossar I
17 Anhang VI - Inhalt Datenträger I

1 Einleitung

Die grundlegende Idee für diese Arbeit entstand im Masterstudiengang Medieninformatik an der Beuth Hochschule für Technik Berlin, in dem ich zum ersten Mal mit dem Arduino Microcontroller in Kontakt kam. Dieser erste Kontakt ereignete sich im Kurs „Interaktive Multimedia-Systeme“, in dessen Verlauf es galt, ein Microcontroller-Projekt zu realisieren. Das damals gewählte Projekt beinhaltete die Aufgabenstellung, über den Microcontroller regelmäßige Twitter[1]-Meldungen zu veröffentlichen, wozu der verwendete Arduino Microcontroller mit dem Internet verbunden und Anfragen an den Twitter-Server gesendet werden mussten. Dies ließ sich zwar umsetzen; für die einfache Aufgabe, eine kurze Zeichenkette an einen Internetserver zu schicken, erschien der Aufwand jedoch unverhältnismäßig hoch.

Ich spiele in meiner Freizeit gerne und ambitioniert Darts und protokolliere dabei meine erzielten Ergebnisse handschriftlich, um Trainingserfolge erkennen und auswerten zu können. Diese Aufgabe ist jedoch sehr mühsam, da innerhalb einer Trainingseinheit leicht mehrere Hundert Würfe absolviert werden. Die Auswertung der notierten Zahlen erfordert zusätzliche Aufwände.

Aus diesem Umstand entstand die Idee, eine elektronische Dartscheibe so zu modifizieren, dass sie an einen Arduino Microcontroller angeschlossen werden könnte. Durch diese Verbindung könnten die erzielten Punkte bei jedem Wurf automatisch über die Verbindung zu einem Computer gespeichert und im Nachhinein ausgewertet werden. Schnell entwickelte sich diese Idee weiter: es könnten auch mehrere elektronische Dartscheiben modifiziert und so die Ergebnisse mehrerer Spieler automatisch aufgezeichnet und ausgewertet werden. Diese Idee würde aber nur dann sinnvoll umgesetzt werden können, wenn nicht jeder Microcontroller an einen anderen Computer angeschlossen wäre, sondern es einen zentralen Ort für die Datenhaltung gäbe – einen Server im Netzwerk oder Internet. Über eine spezielle Software könnten die Ergebnisse einzelner Spieler dann verarbeitet und z.B. untereinander verglichen werden.

Aus der Erfahrung, dass die Anbindung eines Arduino Microcontrollers an das Internet/Netzwerk zwar möglich, der dazu notwendige Aufwand jedoch sehr hoch ist, und dem Wunsch, Hardware (Dartscheibe) über einen solchen Microcontroller an einen Server anzubinden, ergab sich die letztendliche Idee für die vorliegende Arbeit: es soll eine Werkzeugsammlung (*„Toolkit“*) entwickelt werden, mit der nicht nur eine ganz bestimmte, sondern beliebige[2] Hardware in Softwareprojekte eingebunden werden kann. Im Gegensatz zur ursprünglichen Idee soll die Kommunikation bidirektional möglich sein: die Software, in die ein oder mehrere Arduino Microcontroller und an diese angeschlossene Hardware eingebunden sind, soll Daten von diesen empfangen, jedoch auch Daten an diese senden können.

1.1 Gliederung

Zunächst wird in Abschnitt 2 auf das fachliche Umfeld dieser Arbeit eingegangen. Dazu werden Wissensgrundlagen zu Microcontrollern im Allgemeinen und für den Arduino Microcontroller im Speziellen vermittelt, ein Überblick über das verwendete Kommunikationsprotokoll HTTP gegeben, sowie technologische Grundlagen der verwendeten Programmiersprachen und Plattformen angeführt.

Danach wird in Abschnitt 3 auf die aktuelle Marktsituation eingegangen und gezeigt, welche bestehenden Ansätze für die Lösung des dieser Arbeit zugrunde liegenden Problems bereits existieren. Für jeden diskutierten Lösungsansatz werden Vor- und Nachteile genannt. Im Anschluss wird

[1] www.twitter.com
[2] *„beliebig“* bedeutet hier: „kompatibel mit einem Arduino Microcontroller“

beschrieben, welche Eigenschaften der im Zuge dieser Arbeit entwickelten Lösung im Vergleich zu den bestehenden Lösungen von besonderer Bedeutung sind.

Abschnitt 4 stellt das Pflichtenheft für die zu entwickelnde Software dar und beschreibt, welche Funktionen sie bereitstellen muss, mit welchen Daten sie arbeitet und welche qualitativen Anforderungen an sie gestellt werden.

Auf Basis des Pflichtenhefts wird in Abschnitt 5 das Toolkit entworfen. Dabei wird besonders auf die Hervorhebung verwendeter Technologien und Entwurfsstrategien eingegangen und exemplarisch gezeigt, welche Vorteile diese für die Entwicklung der Software bieten. Weiterhin werden eingesetzte Werkzeuge und Bibliotheken, die selbst nicht Bestandteil dieser Arbeit sind, beschrieben. Abschließend werden fachlich besonders interessante Punkte der entwickelten Software detailliert betrachtet, darunter ausgewählte Klassen und das verwendete Kommunikationsprotokoll, sowie bestimmte Verhaltensweisen der Software und der Aufbau der Datenbank erläutert.

Abschnitt 6 beschreibt die Realisierung der Software und geht dabei auf ausgewählte Punkte und Besonderheiten des Toolkits ein. Zusätzlich werden Sicherheitsaspekte diskutiert und qualitätssichernde Maßnahmen beschrieben, die während der Entwicklung eingesetzt wurden.

Die Abschnitte 7 und 8 bilden die Benutzerhandbücher für Anwender der Server-Komponente bzw. der Client-Bibliothek des Toolkits und beschreiben ausführlich, wie diese verwendet, verändert und erweitert werden können. Auch der Einsatz in Verbindung mit einer Drittsoftware wird an dieser Stelle erörtert und entsprechende Codebeispiele werden gegeben. Abschnitt 9 zeigt, wie die im Verlauf dieser Arbeit entstandenen Zusatztools, die für die Verwendung des Toolkits nützlich sind, verwendet werden können.

Abschnitt 10 befasst sich mit der Verwendung des Toolkits und stellt entsprechend das Endbenutzerhandbuch dar.

Abschließend fasst Abschnitt 11 die Ergebnisse dieser Arbeit rückblickend zusammen und gibt darüber hinaus einen Ausblick auf potentielle Weiterentwicklungen und Einsatzszenarien des entwickelten Toolkits.

1.2 Dokumentkonventionen

Dieses Dokument verwendet die folgenden Schriftarten und Schriftstile:

Überschrift der 1. Ebene

Überschrift der 2. Ebene

Überschrift der 3. Ebene

Überschrift der 4. Ebene

Beschriftung

Normaler Text

Erläuterungen, Eigennamen und englische Begriffe

`Klassennamen, Assemblies, Namespaces und Dateinamen im Text`

```
Quellcode (Codewort, Klassenname, String-Literal, Kommentar)
```

1.3 Englische Begriffe und Namen

Aufgrund der technischen Natur dieser Arbeit werden im Text häufig englische Begriffe verwendet. Falls möglich, werden diesen Begriffen die passenden deutschen Artikel vorangestellt, um den Textfluss nicht zu beinträchtigen. Englische Begriffe werden, sollte dies notwendig sein, bei ihrem ersten Auftreten übersetzt.

Ist ein englischer Begriff im fachlichen Umfeld der Arbeit geläufiger als sein deutsches Äquivalent, so wird der englische Begriff verwendet. Dies trifft insbesondere auf die Wörter *„Server“*, *„Client“*, *„Request“* und *„Microcontroller“* zu. Sind der deutsche und der englische Begriff gleich geläufig, werden beide Begriffe synonym verwendet.

1.4 Namensgebung

Viele bekannte Softwareprojekte tragen ein Akronym [1], Backronym [2] oder Apronym[3] als Namen. Bekannte Beispiele sind *PHP* (*„**P**HP: **H**ypertext **P**reprocessor“*) oder *TWAIN* (*„**T**echnology **W**ithout **A**n **I**nteresting **N**ame“*). Auch der Name, der für das im Zuge dieser Arbeit entwickelte Toolkit gewählt wurde, basiert auf einem Akronym: ***ORCHID***.

ORCHID ist ein Homophon [3] auf das Akronym *„ORCIT“ (“**O**nline **R**eal-Life **C**lients **I**nternet **T**oolkit“)*, also *“Werkzeugsammlung für an das Internet angebundene Clients der realen Welt”* und ein Apronym auf das englische Wort für *“Orchidee”*.

1.5 Begriffsdefinitionen

Im Verlauf dieser Arbeit werden verschiedene Begriffe verwendet, die vorab erläutert werden müssen, um das Verständnis nicht zu beeinträchtigen.

Ein ***ORCHID Server*** ist eine Instanz einer Software, die das ORCHID Toolkit (genauer: die Server-Komponente des ORCHID Toolkits, vgl. 4.1) einsetzt; ein ***ORCHID Client*** (oder, wenn der Kontext es zulässt, auch nur *„Client“*) ist entsprechend ein Arduino Microcontroller, der mit einem ***Sketch*** (Programm für einen Arduino Microcontroller, vgl. 2.3.2.1) programmiert wurde, der ebenfalls das ORCHID Toolkit (genauer: die Client-Bibliothek des ORCHID Toolkits, vgl. 4.1) einsetzt.

Eine ***Drittsoftware*** ist eine beliebige Software, die nicht Bestandteil des ORCHID Toolkits ist, es aber einsetzt, um Hardware einzubinden; synonym wird anstelle von *„Drittsoftware“* auch der Begriff *„Software“* verwendet, wenn der Kontext dies zulässt.

Innerhalb eines ORCHID Servers existieren Instanzen von Server-Klassen, die Anfragen von ORCHID Clients bzw. von Drittsoftware entgegennehmen. Zum besseren Verständnis werden diese Instanzen als ***Hardware-Server*** (Server-Instanz, die mit ORCHID Clients kommuniziert) bzw. ***Software-Server*** bezeichnet (Server-Instanz, die mit Drittsoftware kommuniziert). Eine Server-Instanz (also ein Hardware- oder ein Software-Server) nimmt lediglich HTTP-Anfragen entgegen; die eigentliche Verarbeitung der Anfragen geschieht in einer entsprechenden Instanz eines ***Daemons***[4], also eines ***Hardware-Daemons*** bzw. eines ***Software-Daemons***.

Ein ***Aktor*** ist – im Gegensatz zu einem Sensor – ein Bauteil, das eine Aktion ausführen kann, z.B. eine LED (*„**L**ight **E**mitting **D**iode“, deutsch etwa „Licht aussendende Diode“*) oder ein Relais.

---

[3] Ein Apronym ist eine Sonderform des Akronyms, bei dem das Akronym ein bereits existierendes Wort einer natürlichen Sprache bildet (vgl. [1]).

[4] Die Bezeichnung „Daemon“ wurde aus dem Umfeld von UNIX entliehen; dort werden Hintergrundprozesse als „Daemon“ bezeichnet.

2 Fachliches Umfeld

Dieser Abschnitt befasst sich mit dem fachlichen Umfeld der vorliegenden Arbeit, das für das Verständnis der restlichen Abschnitte von großer Bedeutung ist. Zunächst werden grundlegende Informationen zu Microcontrollern gegeben, wobei der Fokus auf dem Arduino Microcontroller liegt. Im Anschluss wird auf das verwendete Kommunikationsprotokoll HTTP eingegangen, bevor die Grundlagen der verwendeten Programmiersprachen und Entwicklungsplattformen erläutert werden.

2.1 Grundlagen Microcontroller

Ein Microcontroller ist ein Halbleiterchip, der in sich mindestens einen Prozessor, Speicher, sowie Ein- und Ausgabemöglichkeiten vereint. Der hauptsächliche Einsatzbereich von Microcontrollern ist deren Verwendung als eingebettetes System (engl. *„embedded system“*) in größeren, komplexen Geräten. Zahlreiche Gegenstände des alltäglichen Lebens werden im Inneren von Microcontrollern gesteuert (die Anzahl der eingebetteten Microcontroller und deren Art variiert dabei von Fall zu Fall), darunter u.a. Waschmaschinen, Handys, MP3-Player, elektronische Bilderrahmen, Messgeräte, Taschenrechner oder auch Eingabegeräte für Computer.

2.1.1 Microcontroller allgemein

Dadurch, dass Microcontroller die grundlegenden Funktionalitäten eines Computers in einem Chip vereinen, bieten sich viele Vorteile, die der Hauptgrund für ihre enorme Verbreitung sind. Diese Vorteile sollen im Folgenden erläutert werden.

2.1.1.1 Vielfalt durch Programmierbarkeit

Microcontroller können programmiert werden, um bestimmte Aufgaben durchzuführen. So ist es grundsätzlich denkbar, dass ein Microcontroller, der in einer Kaffeemaschine verbaut ist, die Wassertemperatur über einen Temperatursensor überwacht, während ein zweiter Microcontroller des gleichen Typs, der jedoch anders programmiert wurde, in einem Handy dafür verantwortlich ist, dass bei einem eingehenden Anruf das Display aufleuchtet. Ohne Microcontroller müssten für beide Aufgaben komplexe, nicht in Geräten mit anderen Aufgaben wiederverwendbare Schaltungen entwickelt werden.

2.1.1.2 Lösen komplexer Aufgaben

Genau wie ein vollwertiger Computer können Microcontroller – wie in 2.1.1.1 beschrieben – auf unterschiedliche Art und Weise programmiert werden. Durch diese Eigenschaft können sie – ebenfalls genau wie Computer – auch sehr komplexe Aufgaben übernehmen, die ansonsten nur durch sehr aufwändige Schaltungen oder in dieser Form gar nicht realisierbar wären. Ein Beispiel für diese vorteilhafte Eigenschaft sind MP3-Player. In ihnen sind Microcontroller verbaut, die neben zahlreichen anderen Funktionen (z.B. Interpretation eines Tastendrucks und entsprechende Reaktion darauf) auch das Decodieren von MP3-Dateien übernehmen und somit das Abspielen einer solchen Datei ermöglichen.

2.1.1.3 Effizienz durch Spezialisierung

Microcontroller sind keine vollständigen Computer. Sie arbeiten bedeutend langsamer als jeder durchschnittliche Computer (vgl. 2.1.2.1) und sind auch in anderen Bereichen (z.B. Speicher, vgl. 2.1.2.2) einem Computer unterlegen. Im Gegenzug sind sie aber für den Einsatz als eingebettete Systeme ausgezeichnet geeignet, da sie einerseits sehr klein sind, andererseits sehr wenig Energie brauchen, um zu funktionieren. Die geringere Leistung gegenüber Computern ist durch die Spezialisierung auf die Verwendung als eingebettetes System in größeren Geräten und Maschinen nicht als Nachteil zu interpretieren, da für diese Aufgabe keine hohe Rechenleistung oder Speicherkapazität notwendig ist (Maschinen und Geräte, die diese Anforderungen haben (z.B. Industrieroboter), verfügen meist zusätzlich über integrierte oder angeschlossene vollwertige

Computer). Microcontroller können also effizient als Bauteile für komplexe Geräte und Maschinen eingesetzt werden, ohne diese negativ zu beeinflussen (ein integrierter Computer dagegen verbraucht viel Energie und produziert eine große Menge an Wärme, die abgeführt werden muss).

Microcontroller sind zudem sehr kosteneffizient. Verbreitete Modelle können – auch in kleinen Stückzahlen – zu Preisen von ca. 1-20 EUR im Einzelhandel erworben werden. Der durchschnittliche Preis liegt dabei unter 10 EUR. Die Preise, die von Unternehmen, die Microcontroller für die Entwicklung eigener Produkte einsetzen, an Großhändler oder Hersteller gezahlt werden müssen, sind entsprechend geringer. Aktuelle Prozessoren für Computer kosten (auch in großen Stückzahlen) wesentlich mehr und benötigen zudem kostenintensivere zusätzliche Bauteile.

Aktuelle Einzelhandelspreise für die Microcontroller, die in der Arduino Produktreihe verbaut werden (vgl. 2.1.2.1) können der folgenden Tabelle entnommen werden[5].

Microcontroller	Anbieter	Einzelpreis
Atmel ATmega168	Conrad Electronic SE[6]	4,37€ - 4,94€
	RS Components[7]	1,92€ - 5,22€
	Reichelt Elektronik[8]	3,65€
Atmel ATmega328	Conrad Electronic SE	5,41€
	RS Components	4,11€ - 5,60€
	Reichelt Elektronik	*nicht verfügbar*

Tabelle 1 - Preisübersicht ATmega Microcontroller

2.1.1.4 Flexibilität durch Komponenten

Wie bereits in 2.1.1.1 beschrieben, können Microcontroller durch unterschiedliche Programmierung an verschiedenste Aufgabenbereiche angepasst werden. Diese Anpassungsfähigkeit beruht jedoch nicht allein auf der Programmierbarkeit eines Microcontrollers, sondern auch darauf, dass neben dem Prozessor und dem Speicher auch Ein- und Ausgabemöglichkeiten auf einem Microcontroller vorhanden sind. Über diese können verschiedenste Sensoren und Aktoren an einen Microcontroller angeschlossen werden, mit denen Daten aus der Umgebung gemessen (Sensoren) und diese auch manipuliert werden kann (Aktoren). Der gleiche Microcontroller kann durch seine Programmierung z.B. über den Wert eines angeschlossenen Lichtsensors entscheiden, ob eine Lampe eingeschaltet werden soll, oder mittels eines angeschlossenen LC-Displays eine Meldung ausgeben.

2.1.2 Die Arduino Microcontroller-Plattform

In dieser Arbeit wird ein Toolkit entwickelt, das speziell auf die Arduino Microcontroller-Plattform ausgerichtet ist, deren technische Besonderheiten in den folgenden Abschnitten erläutert werden. Zunächst soll jedoch beschrieben werden, wobei es sich bei der Arduino Microcontroller-Plattform handelt.

Die Arduino-Plattform besteht aus zwei Teilen: dem eigentlichen Microcontroller-Board (also einer Platine, die neben dem Microcontroller auch weitere Bauteile wie Eingangs- und Ausgangs-Pins, Status-LEDs, Peripherie-Anschlüsse, etc. trägt), häufig als *„I/O-Board“* oder einfach *„Arduino“* bezeichnet, und einer Entwicklungsumgebung (IDE), die ebenfalls den Namen *„Arduino“* trägt. Die Entwicklungsumgebung und die zugehörige Programmiersprache (auch diese trägt den Namen *„Arduino“*) werden in 2.3.2 beschrieben.

[5] Stand: 10.09.2011, 20:00 Uhr. Es handelt sich nicht um eine Auflistung aller Anbieter.
[6] http://www.conrad.de
[7] http://de.rs-online.com/
[8] http://www.reichelt.de

Es gibt zahlreiche verschiedene Arduino Microcontroller-Boards mit unterschiedlichen Spezifikationen (Details sind Abschnitt 2.1.2.1 zu entnehmen); der Fokus dieser Arbeit liegt auf dem Arduino *„Duemilanove“*[9], der in der folgenden Abbildung dargestellt ist.

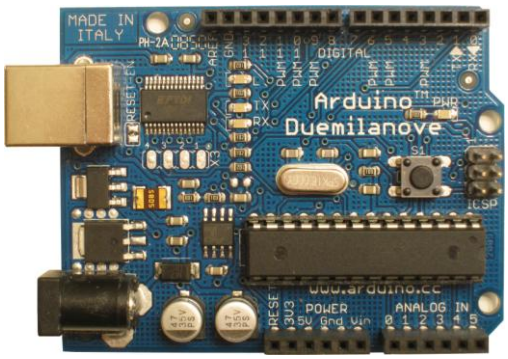

Abbildung 1 - Arduino "Duemilanove"

Eine besondere Eigenschaft der Arduino Microcontroller-Plattform ist ihre Veröffentlichung als Open-Source-Projekt. Sowohl die Software, als auch die Diagramme für die Hardware sind quelloffen und können verändert oder selbst implementiert bzw. gebaut werden. Dieser Aspekt ist meiner Meinung nach einer der Hauptfaktoren für die enorme Beliebtheit, der sich die Arduino-Plattform erfreuen kann; zusätzlich wird es Entwicklern so erheblich erleichtert, ihre eigenen Erweiterungen für die Plattform zu entwickeln und der Öffentlichkeit zur Verfügung zu stellen (vgl. 2.1.2.3).

Ein weiterer Grund für die Beliebtheit der Plattform ist deren Ausrichtung auf die nahezu spielerische Entwicklung von Prototypen. Die Philosophie der Plattform ist dem Internetauftritt des Herstellers [4] zu entnehmen: *„Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments.” (deutsch: “Arduino ist eine Open-Source Elektronik-Prototypen-Plattform, die auf flexibler, einfach zu nutzender Hard- und Software basiert. Sie ist für Künstler, Designer, Hobbyisten und jeden, der daran interessiert ist, interaktive Objekte und Umgebungen zu kreieren, gedacht.”).*

2.1.2.1 Microcontroller

Die Arduino-Plattform verwendet 8-Bit-Microcontroller der Firma Atmel aus der Microcontroller-Familie *„AVR“*. Die Microcontroller der AVR-Familie verfügen über RISC-Prozessoren (*„Reduced Instruction Set Computer“, deutsch: „Rechner mit reduziertem Befehlssatz“*). Die RISC-Architektur bietet den Vorteil, dass alle Befehle für einen RISC-Prozessor innerhalb eines Prozessortakts abgearbeitet werden können, was wiederum Geschwindigkeitsvorteile mit sich bringt.

Es gibt verschiedene Arduino I/O-Boards, die mit unterschiedlichen Atmel-AVR-Microcontrollern ausgestattet sind. Eine detaillierte Liste kann der Herstellerseite entnommen werden (siehe [5]). Für diese Arbeit werden Arduino Microcontroller-Boards des Typs *Duemilanove* verwendet. Sie verfügen über ATmega168-Microcontroller (es gibt auch Modelle, in denen ein ATmega328-Microcontroller verbaut ist) aus der Atmel-AVR-Familie und arbeiten mit einer Taktfrequenz von 16 MHz (vgl. [6], S. 18, und [7], technische Details auf der Herstellerseite, siehe [8]).

2.1.2.2 Speicher

Arduino *Duemilanove* Boards verfügen über 32 Kilobyte Flash-Speicher für Programme und 2 Kilobyte RAM (vgl. [7]). Andere Arduino-Boards verfügen über mehr oder weniger Speicher (vgl.

[9] Italienisch für „zweitausendneun“ – in diesem Jahr wurde der Duemilanove veröffentlicht.

[5]), was zu Einschränkungen im Zusammenhang mit dem hier entwickelten Toolkit führen kann (vgl. 6.6.4).

2.1.2.3 Erweiterung mit Shields

Eine der Besonderheiten von Arduino I/O-Boards, die diese von anderen Microcontroller-Plattformen unterscheidet, ist die Fähigkeit, über sog. *„Shields“ (deutsch: „Schilde“)* nahezu beliebig erweiterbar zu sein. Ein Shield ist eine Erweiterung für ein I/O-Board, die auf das Board gesteckt werden kann, daher auch der Name. Mehrere Shields können übereinander auf ein I/O-Board gestapelt werden, um die Funktionen, die sie mitbringen, miteinander zu kombinieren.

Es gibt unterschiedlichste Arten von Shields, die verschiedene Erweiterungen mit sich bringen. Das wohl bekannteste Shield ist das sog. *„Ethernet Shield“* (siehe [9]), das es einem Arduino I/O-Board erlaubt, sich über ein Netzwerkkabel mit dem Internet oder Netzwerk zu verbinden. Weitere Shields erweitern ein I/O-Board z.B. um die Fähigkeit, GPS-Signale zu empfangen (GPS-Shield[10]), einen Touchscreen nutzen zu können (2.8‘‘ TFT Touch Shield[11]), oder sogar Videospiele über einen VGA-Monitor darstellen zu können (Gameduino Shield[12]). Eine sehr umfangreiche Liste verfügbarer Arduino-Shields kann der *„Arduino Shield List“* von Jonathan Oxer[13] entnommen werden.

2.1.2.4 Erleichterte Programmierung

Wie bereits beschrieben, umfasst die Arduino Microcontroller-Plattform auch eine *IDE (**I**ntegrated **D**evelopment **E**nvironment, deutsch etwa: „Integrierte Entwicklungsumgebung“)* und eine Programmiersprache, die für die Programmierung von Arduino I/O-Boards ausgelegt sind. Im Vergleich zu anderen Microcontrollern kann ein Arduino I/O-Board sehr viel leichter programmiert werden.

Microcontroller zu programmieren ist ein aufwändiger Prozess, da in den meisten Fällen als Programmiersprache C/C++ oder ein davon abgeleiteter Dialekt zum Einsatz kommen muss. Nicht nur tiefgreifende Kenntnisse dieser Sprachen, sowie ein fundiertes technisches Verständnis der eingesetzten Microcontroller sind erforderlich, sondern oft auch der Einsatz spezieller Hardware, die die Programmierung des Microcontrollers überhaupt erst möglich macht (sog. *„Programmer“*). Im Gegensatz dazu ermöglichen es die Arduino-IDE und die Arduino-Programmiersprache auch unerfahrenen Programmierern, einen Microcontroller zu programmieren. Zahlreiche Beispiele, die zum Lieferumfang der IDE gehören, und zahllose Ressourcen im Internet erleichtern den Einstieg zusätzlich.

Die Programmierung eines Arduino I/O-Boards für Einsteiger wird erleichtert, indem die Arduino-Programmiersprache die aus der Verwendung von C/C++ stammenden Probleme *„durch viele hilfreiche Konstrukte vor dem Benutzer versteckt*“ ([6], S. 5). Dennoch können die fortgeschrittenen Konzepte, die diese Programmiersprachen anbieten, auf Wunsch genutzt werden, wodurch eine größere Flexibilität und eine Erweiterung der Möglichkeiten für Entwickler erreicht werden.

2.1.2.5 Weitere Eigenschaften

Ein Arduino I/O-Board verfügt über – je nach Board unterschiedlich viele – Ein- und Ausgänge (*„I/O-Pins“*). Die genaue Zahl für jeden Board-Typ kann der Spezifikation auf der Herstellerseite entnommen werden (siehe [5]). Eine der Besonderheiten der Arduino Microcontroller-Plattform ist die Verfügbarkeit von zwei Arten von I/O-Pins: es gibt *digitale* Pins, die sowohl als Eingang, als auch als Ausgang konfiguriert werden können, und *analoge* Pins, die ausschließlich als Eingang dienen.

[10] http://www.adafruit.com/products/98
[11] http://www.adafruit.com/products/376
[12] http://www.adafruit.com/products/384
[13] http://www.shieldlist.org/

Weitere Pins auf jedem I/O-Board dienen u.a. der Spannungsversorgung angeschlossener Komponenten.

Der Arduino *Duemilanove* verfügt über 14 digitale I/O-Pins (0-13) und 6 analoge Eingänge (0-5); Details sind der Spezifikation des Duemilanove auf der Herstellerseite zu entnehmen (siehe [7]).

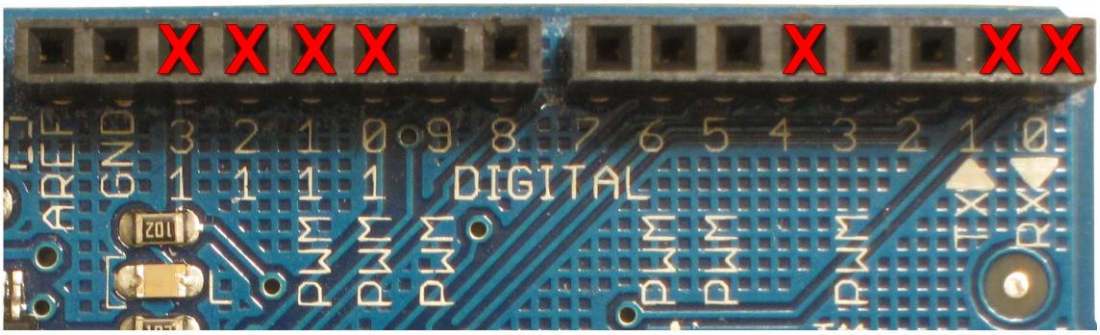

Abbildung 2 - Digitale I/O-Pins des Arduino Duemilanove

Die obige Abbildung zeigt die digitalen I/O-Pins des Arduino Duemilanove; rot markierte Pins können unter bestimmten Umständen für reguläre Eingabe/Ausgabe-Aufgaben nicht verwendet werden: die digitalen Pins 0 und 1 werden intern vom I/O-Board für die serielle Kommunikation mit einem Computer genutzt (vgl. [6], S. 20). Die Pins 10 bis 13 werden dafür verwendet, mit weiterer Peripherie (nicht Sensoren und Aktoren) wie beispielsweise einem Shield zu kommunizieren und können, falls solche Peripherie verwendet wird, ebenfalls nicht regulär genutzt werden (vgl. [6], S. 20 und [9]). Ein Sonderfall ist Pin 4: dieser kann bei Verwendung der neuen Version des Ethernet Shield mit integriertem Micro-SD Kartenslot nicht genutzt werden; solche Shields nutzen Pin 4 zur Kommunikation mit dem Kartenslot (siehe [9]).

Für diese Arbeit kommen sowohl die serielle Kommunikation zwischen I/O-Board und Computer, als auch ein Ethernet Shield mit integriertem Micro-SD Kartenslot zum Einsatz. Die in Abbildung 2 markierten digitalen I/O-Pins stehen also nicht zur Verfügung.

2.2 Grundlagen HTTP

Das im Rahmen dieser Arbeit entwickelte ORCHID Toolkit versetzt Hard- und Software in die Lage, über eine zentrale Stelle im Netzwerk oder Internet (einen Server) miteinander zu kommunizieren. Sowohl Hardware (Arduino Microcontroller), als auch Software (Drittsoftware) übernehmen also die Rolle eines Clients, der mit einem Server kommuniziert.

Die Kommunikation zwischen einem Client und einem Server basiert auf dem ***H**ypertext **T**ransfer **P**rotocol (HTTP)* in der Version 1.1. Die Grundlagen dieses Protokolls werden im Folgenden erläutert.

2.2.1 Allgemeines

HTTP ist ein Netzwerkdatentransferprotokoll, also ein Protokoll für den Austausch von beliebigen Daten zwischen zwei Kommunikationspartnern, die an das gleiche Netzwerk angeschlossen sind. Es wurde in den 1990er Jahren als Grundlage des heute allgegenwärtigen Internets entwickelt und im Mai 1996 als RFC 1945 in der Version 1.0 veröffentlicht. Im Juni 1999 wurde mit RFC 2616 der auch heute noch verwendete Nachfolger HTTP/1.1 veröffentlicht. Die vollständigen Spezifikationen von RFC 1945 bzw. RFC 2616 können [10] bzw. [11] entnommen werden und befinden sich zusätzlich auf dem dieser Arbeit beiliegenden Datenträger.

Der eigentliche Zweck und das am weitesten verbreitete Einsatzgebiet des Protokolls ist die Übertragung von *HTML*-Dokumenten (***H**ypertext **M**arkup **L**anguage*) in einem Netzwerk, bzw. über das Internet; HTTP ist aber in der Lage, beliebige Daten zu übermitteln.

HTTP ist ein anwendungsorientiertes Protokoll, das auf den Schichten 5 bis 7 des ISO/OSI-Referenzmodells arbeitet und auf ***T**ransmission **C**ontrol **P**rotocol* (TCP)[14] als Datentransferprotokoll aufsetzt.

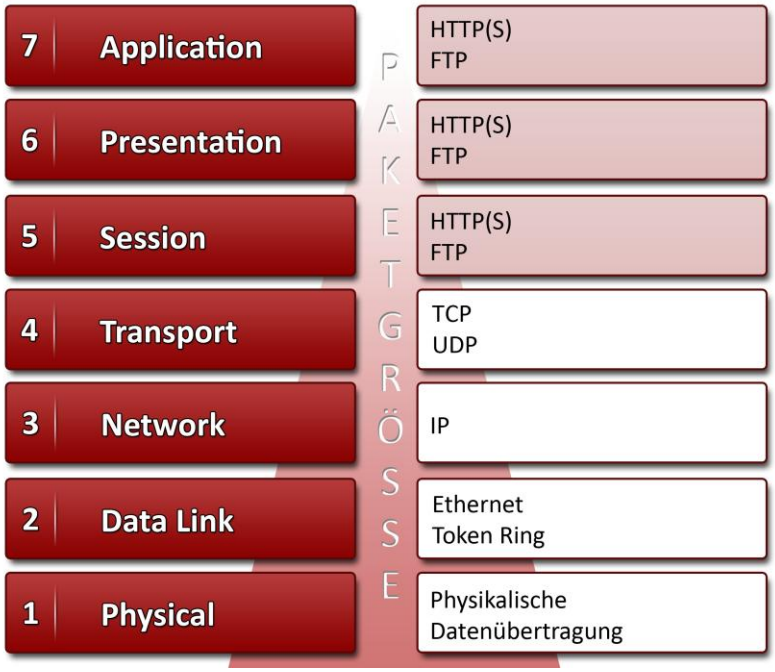

Abbildung 3 - ISO/OSI-Referenzmodell

Die obige Abbildung 3 zeigt die Schichten des ISO/OSI-Referenzmodells (links) und auszugsweise die Protokolle, die auf den jeweiligen Schichten arbeiten (rechts).

[14] RFC 793, siehe http://tools.ietf.org/html/rfc793

Da HTTP in den höheren Schichten 5 bis 7 arbeitet, können Anwendungsentwickler das Protokoll leicht für eigene Projekte verwenden, ohne tiefer in ein System eingreifen zu müssen. Die unteren Schichten erfordern hingegen einen weitaus größeren Aufwand, um Änderungen zu implementieren (je niedriger die Nummer einer Schicht ist, desto stärker ist ihr Bezug zu Hardware).

HTTP ist ein zustandsloses Protokoll, d.h. zwischen je zwei Übertragungen gibt es per Definition keine Zusammenhänge (dies wird erst durch Erweiterungen des Protokolls auf Anwendungsebene möglich, ist aber für diese Arbeit nicht von Bedeutung). Es werden zwei Arten von Nachrichten (*Messages*) unterschieden: Anfragen (*Requests*) und Antworten (*Responses*). Aufgrund der Zustandslosigkeit des Protokolls muss ein Kommunikationspartner in der Lage sein, eine Antwort einer Anfrage zuzuordnen und umgekehrt.

Eine HTTP-Nachricht besteht immer aus einem Nachrichtenkopf (*Header*) und einem optionalen Nachrichtenkörper (*Body*).

2.2.2 Anfragen (Requests)

Ein Kommunikationspartner, der eine Anfrage stellt, wird zu einem Client. Der Partner, der die Anfrage entgegennimmt, wird entsprechend zu einem Server.

Der Client erzeugt eine Anfrage, die nach einem bestimmten Schema aufgebaut ist, das in Listing 1 gezeigt wird. Pflichtangaben mit variablen Werten sind im Listing in spitze Klammern eingeschlossen, optionale Angaben mit variablen Werten in eckige Klammern. Angaben ohne Klammern sind Pflichtangaben mit festem Wert („`CRLF`“ steht für „***C**arriage **R**eturn **L**ine **F**eed*“, also einen Zeilenumbruch, der aus den Steuerzeichen „*Carriage Return*“ und „*Line Feed*“ besteht, die häufig als „`\r\n`“ abgekürzt werden).

```
<METHOD> <URI> <HTTP-VERSION> CRLF
[HEADER-FIELD 1] CRLF
[HEADER-FIELD 2] CRLF
...
[HEADER-FIELD n-1] CRLF
[HEADER-FIELD n] CRLF
CRLF
[MESSAGE BODY]
```

Listing 1 - Struktur einer HTTP-Nachricht

Die Reihenfolge, in der die im obigen Listing gezeigten Angaben gemacht werden, ist nur für die erste Zeile der Anfrage und für die Trennung zwischen Header-Feldern und Message-Body von Bedeutung. Header-Felder können in beliebiger Reihenfolge übertragen werden.

2.2.2.1 HTTP-Methoden

HTTP unterscheidet verschiedene Arten von Anfragen. Die Unterscheidung beruht dabei hauptsächlich auf der übermittelten Methode (es handelt sich hier nicht um eine Methode im Sinne der objektorientierten Programmierung).

Die folgenden Methoden sind Teil des Standards HTTP/1.1 (siehe [11], Sektion 9.2 ff.). Die Reihenfolge der Methoden wurde RFC 2616 entnommen.

Methode	Einsatz
OPTIONS	Erlaubt es einem Client, die für eine angegebene Ressource verfügbaren Optionen von einem Server abzurufen.
GET	Fordert eine durch einen URI[15] spezifizierte Ressource von einem Server an. Der Server übermittelt die Ressource im Message-Body der Antwort.
HEAD	Identisch mit GET, jedoch ist der Server nicht verpflichtet, einen Message-Body zu übertragen. Wird hauptsächlich eingesetzt, um zu prüfen, ob ein URI gültig ist.
POST	Erlaubt es einem Client, Daten auf einen Server zu laden. Die Verwendung der Daten obliegt jedoch dem Server. Im Gegensatz zu PUT werden unter Verwendung von POST Daten als neue Teilmenge einer bestehenden Ressource angesehen.
PUT	Erlaubt es einem Client (wie POST), Daten auf einen Server zu laden. Im Gegensatz zu POST werden diese Daten jedoch nicht als Teilmenge einer existierenden Ressource betrachtet, sondern als neue, eigene Ressource oder ggf. als Aktualisierung einer solchen.
DELETE	Über die Methode DELETE kann ein Client einen Server dazu auffordern, eine durch einen URI spezifizierte Ressource zu löschen.
TRACE	Erlaubt es einem Client, die Veränderungen eines Requests durch andere Hosts (falls solche existieren) zu überprüfen.
CONNECT	CONNECT ist eine für Proxy-Server reservierte Methode und wird an dieser Stelle nicht weiter behandelt.

Tabelle 2 - HTTP-Methoden

Es existieren noch weitere, nicht im Standard enthaltene Methoden (und es ist leicht möglich, weitere zu implementieren). Die folgende Abbildung 4 zeigt ein Bildschirmfoto der Software „*Fiddler*“[16], mit der HTTP-Anfragen an einen Server manuell aufgebaut werden können. Die abgebildete Auswahlliste enthält verschiedene weitere HTTP-Methoden.

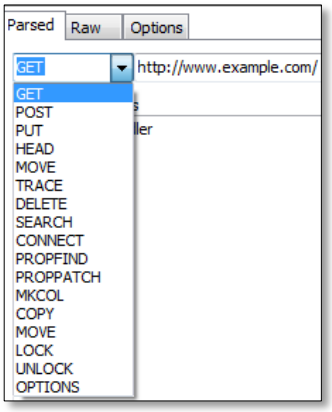

Abbildung 4 - Screenshot "Fiddler"

Wie aus Listing 1 ersichtlich wird, muss die Methode, die ein Client für eine Anfrage verwenden will, stets in der ersten Zeile eines Requests enthalten sein. Das folgende Listing 2 zeigt beispielhaft die ersten Zeilen für die in Tabelle 2 gezeigten HTTP-Methoden.

[15] *„**U**niform **R**esource **I**dentifier“, deutsch etwa: „Einheitlicher Ressourcen-Bezeichner“*
[16] http://www.fiddler2.com/

```
OPTIONS <URI> HTTP/1.1 CRLF
GET <URI> HTTP/1.1 CRLF
HEAD <URI> HTTP/1.1 CRLF
POST <URI> HTTP/1.1 CRLF
PUT <URI> HTTP/1.1 CRLF
DELETE <URI> HTTP/1.1 CRLF
TRACE <URI> HTTP/1.1 CRLF
CONNECT <URI> HTTP/1.1 CRLF
```

Listing 2 - Übersicht HTTP-Methoden

Es handelt sich bei den im obigen Listing gezeigten Zeilen nur um die erste Zeile einer vollständigen HTTP-Anfrage, Header-Felder und Message-Body fehlen; die Anfragen sind entsprechend nicht gültig.

2.2.3 HTTP-Header

Der Standard HTTP/1.1 kennt 47 verschiedene Header-Felder (vgl. [11], Sektion 14), die unterschiedliche Metadaten über die im Message-Body übertragenen Daten enthalten können. Auch ohne die Übertragung eines Message-Bodys können Header-Felder verwendet werden.

HTTP-Header werden als Schlüssel-Wert-Paare (Key-Value-Pairs) übertragen und haben immer die im folgenden Listing gezeigte Form.

```
<KEY>:<VALUE> CRLF
```

Listing 3 - Struktur eines HTTP-Headers

Eine Auflistung aller Header-Felder, die im HTTP-Standard bekannt sind, würde den Rahmen dieser Arbeit sprengen. Die folgende Tabelle enthält daher ausgesuchte Header-Felder und gibt deren Verwendungszweck an.

Header-Feld	Verwendungszweck
Content-Length	Gibt die Länge (in Bytes) der Nachricht bzw. übertragenen Daten im Message-Body an.
Host	Gibt den Server an, von dem eine Ressource abgerufen werden soll.
Referer	Wenn ein Host einen Client auf eine Ressource auf einem anderen Server weiterleitet, wird der weiterleitende Host in diesem Header-Feld benannt.
User-Agent[17]	Ein Client übermittelt einem Server über dieses Header-Feld seine eigene Identifikation, aus der der Server Schlüsse ziehen kann (z.B. bei der Auslieferung von Websites).

Tabelle 3 - Ausgewählte Header-Felder

Es ist möglich, Header-Felder zu übertragen, die nicht Teil des Standards sind. In diesem Fall müssen sich Client und Server jedoch auf ein gemeinsames Protokoll einigen, das auf HTTP aufbaut, da ansonsten Header, die einem Server unbekannt sind, von diesem ignoriert werden.

2.2.4 HTTP Status-Codes

Der HTTP-Standard unterscheidet fünf Kategorien von Status-Codes (auch *„Response-Codes“* genannt), die in der folgenden Tabelle dargestellt sind. Jeder Status-Code wird durch eine dreistellige Zahl repräsentiert.

[17] http://www.useragentstring.com/

Kategorie	Code	Bedeutung
Informational	**1XX**	Wird für provisorische Antworten verwendet, z.B. wenn die eigentliche Antwort noch nicht bereit ist.
Successful	**2XX**	Wenn eine Client-Anfrage erfolgreich bearbeitet wurde, wird eine Antwort aus dieser Kategorie übermittelt.
Redirection	**3XX**	Kann eine angeforderte Ressource nicht ohne eine Umleitung an einen anderen Server ermittelt werden, werden Antworten dieser Kategorie versendet.
Client Error	**4XX**	Wird verwendet, wenn ein Client eine fehlerhafte oder unvollständige Anfrage an einen Server gesendet hat.
Server Error	**5XX**	Diese Kategorie wird für Antworten verwendet, die einen Fehler des Servers anzeigen.

Tabelle 4 - HTTP Status-Codes

Wie die Methode, die ein Client in einer HTTP-Anfrage in der ersten Zeile der versendeten Nachricht benennen muss, wird auch der Status-Code einer Antwort als erste Information innerhalb einer Server-Antwort erwartet. Listing 4 zeigt beispielhaft die Antwort eines Servers auf eine erfolgreiche GET-Anfrage.

```
HTTP/1.1 200 OK CRLF
Date: Sun, 04 Sep 2011 19:31:57 GMT CRLF
Content-Type: text/html; charset=ISO-8859-1 CRLF
Content-Length: <length> CRLF
CRLF
<Message Body>
```

Listing 4 - Serverantwort "200 OK"

Eine nicht erfolgreiche Anfrage kann auf verschiedene Arten beantwortet werden. Der übermittelte Status-Code hängt dabei von den Gründen ab, warum die Anfrage nicht erfolgreich bearbeitet werden konnte.

Eine Anfrage, die eine nicht vorhandene Ressource auf einem Server abrufen möchte, wird z.B. mit Statuscode 404 *(„Not Found“)* beantwortet, wie das folgende Listing zeigt.

```
HTTP/1.1 404 Not Found CRLF
Date: Sun, 04 Sep 2011 19:41:54 GMT CRLF
Content-Length: <n> CRLF
Content-Type: text/html; charset=iso-8859-1 CRLF
CRLF
<Message Body>
```

Listing 5 - Serverantwort "404 Not Found"

Eine vollständige Liste aller HTTP Status-Codes kann [11], Sektion 10, entnommen werden.

2.2.5 Verwendung von HTTP im ORCHID Toolkit

Das ORCHID Toolkit stellt keine Implementierung des HTTP-Standards dar, sondern verwendet das Protokoll lediglich, um Daten zwischen Clients und Servern auszutauschen. Zusätzlich zu dieser Einschränkung ist das Toolkit nicht dafür ausgelegt, Anfragen von Webbrowsern entgegenzunehmen und zu verarbeiten. Entsprechend sind Abweichungen vom HTTP-Standard und die Einführung neuer Header-Felder unproblematisch (vgl. 5.8).

ORCHID Clients identifizieren sich über einen entsprechenden Wert im Header-Feld *„User Agent“*. Beinhaltet dieses Feld einen Wert, der einen Client nicht als gültigen ORCHID Client ausweist, werden dessen Anfragen abgelehnt, wie in Abschnitt 5.8 ausführlich beschrieben wird.

2.3 Technologische Grundlagen

Im Folgenden werden technologische Grundbegriffe der verwendeten Programmiersprachen und Entwicklungsplattformen erläutert. Eine detaillierte Beschreibung der Möglichkeiten, die diese Technologien bieten, kann Abschnitt 5.4 entnommen werden.

2.3.1 Mono

Das Mono Framework[18] ist eine von der Firma Novell[19] unterstützte, quelloffene Entwicklungsplattform. Mono basiert auf dem von der Microsoft Corporation[20] entwickelten .NET Framework[21], ist aber im Gegensatz dazu nicht nur für Microsoft Windows, sondern auch für Linux und MacOS verfügbar.

Mono erlaubt es, Anwendungen in mehreren Programmiersprachen (darunter insbesondere C#) zu entwickeln, die dann sowohl unter Microsoft Windows, als auch Linux und MacOS ausführbar sind – vorausgesetzt, das Mono Framework ist auf dem ausführenden System installiert. Eine aktuelle Liste aller von Mono unterstützen Sprachen kann von der Herstellerwebsite [12] abgerufen werden.

Zur Ausführung von Mono-Programmen wird eine Laufzeitumgebung benötigt. Wie in Monos Vorbild, dem Microsoft .NET Framework, wird Programmcode vom Compiler nicht direkt in Maschinencode übersetzt, sondern in eine Form von Zwischencode, den sog. ***C**ommon **I**ntermediate **L**anguage **C**ode (CIL-Code)*. CIL-Code ist nicht direkt ausführbar (ähnlich wie *Java-Bytecode*, der von der *Java Virtual Machine (JVM)* ausgeführt werden muss), sondern kann nur von einer virtuellen Maschine, die die sog. ***C**ommon **L**anguage **I**nfrastructure (CLI)* implementiert, ausgeführt werden. Im .NET Framework von Microsoft wird diese Aufgabe von der ***C**ommon **L**anguage **R**untime (CLR)* übernommen; Mono verfügt zu diesem Zweck über die *Mono Runtime*.

Wird – wie in vielen anderen Programmiersprachen üblich – keine virtuelle Maschine zur Ausführung verwendet, muss ein Programm für jedes Betriebssystem, auf dem es lauffähig sein soll, einzeln kompiliert werden, da der Compiler es speziell auf die Architektur des übersetzenden Computers abgestimmt in Maschinensprache überführt. Auch eine virtuelle Maschine übersetzt ein Programm in Maschinencode, der vom ausführenden Computer verarbeitet werden kann; dies geschieht jedoch über einen ***J**ust-**I**n-**T**ime-Compiler (JIT-Compiler)*, der bereits in eine Zwischensprache (hier: CIL) übersetzten Code zur Laufzeit des Programms in ausführbaren Maschinencode übersetzt. So können Binärdateien verteilt und unter verschiedenen Betriebssystemen ausgeführt werden, ohne neu kompiliert werden zu müssen.

2.3.1.1 Solutions und Projekte

Als plattformunabhängige Portierung des .NET Framework arbeitet auch Mono mit den daraus bekannten *Solutions* (dt. *„Lösungen“*) und *Projekten*. Bei einem Projekt handelt es sich um eine Zusammenfassung von Quellcodedateien, Ressourcen und Referenzen auf Bibliotheken, die zusammen ein Programm oder eine Bibliothek bilden.

Eine Solution ist eine Sammlung von Projekten. Es ist nicht erforderlich, dass alle Projekte innerhalb einer Solution vom gleichen Typ sind; es ist auch möglich, Projekte in verschiedenen Sprachen zu

[18] http://www.mono-project.com
[19] http://www.novell.com
[20] http://www.microsoft.com
[21] http://www.microsoft.com/net/

entwickeln und sie in einer Solution zu verwalten, solange die verwendeten Sprachen von Mono unterstützt werden.

2.3.1.2 Assemblies

Eine *Assembly* (dt. etwa *„Gruppe“* oder *„Verband“*) ist ein in die binäre Form überführtes Projekt. Assemblies können ausführbare Programme oder Bibliotheken sein. Bibliotheken können – im Gegensatz zu Programmen – nicht direkt von einer die CLI implementierenden virtuellen Maschine ausgeführt werden, sondern stellen ihre Funktionen anderen Programmen zur Verfügung.

2.3.1.3 Namespaces

Namespaces (dt. *„Namensräume“*) ermöglichen es, zusammengehörige Klassen innerhalb eines Projekts in logische Einheiten zu unterteilen, ohne das Projekt aufteilen zu müssen. In Java sind Namespaces als *Pakete* bekannt.

2.3.2 Arduino

Dieser Abschnitt beschreibt die Programmiersprache *Arduino* und geht auf deren Besonderheiten ein.

Arduino basiert auf *Wiring*[22], einer Sprache, die für die Programmierung eines gleichnamigen Microcontroller-Boards entwickelt wurde und ihrerseits auf *Processing*[23] basiert. Processing wurde hauptsächlich für die künstlerische Anwendung der Programmierung, also das Erzeugen von Bildern und Animationen, entworfen.

Processing, Wiring und Arduino ähneln sich äußerlich sehr. Die Syntax der Sprachen ist vergleichbar mit der von C/C++, Java und C#. Wie auch Wiring basiert Arduino auf C/C++ (Processing basiert auf Java).

Durch die große Ähnlichkeit der Syntax von Arduino mit der bekannter Programmiersprachen wie C/C++, Java und C# fällt einer großen Zahl von Programmierern der Einstieg in die Sprache sehr leicht. Um diese Tatsache zu verdeutlichen, wird im folgenden Listing eine Methode definiert, die zwei Ganzzahlen addiert. Diese Methode kann ohne Anpassungen u.a. in Arduino, Processing, Wiring, Java, C/C++ und C# verwendet werden.

```
01 int add(int a, int b)
02 {
03     return a + b;
04 }
```

Listing 6 - Beispielmethode in mehreren Programmiersprachen

2.3.2.1 Sketches

Ein Programm, das für die Ausführung durch einen Arduino-Microcontroller entwickelt wurde, wird *Sketch (deutsch: „Konzept“ oder „Skizze“)* genannt. Ein Sketch besteht dabei aus einer oder mehreren Dateien, in denen der Quellcode des Programms enthalten ist; benötigte Bibliotheken können in Sketches eingebunden werden, falls der Entwickler dies wünscht.

2.3.2.2 Bibliotheken

Die Arduino-IDE ermöglicht die Einbindung von Bibliotheken (auch *„Library“* genannt) in einen Sketch. Eine Bibliothek ist in C++ geschriebener Code, der vom eigentlichen Sketch unabhängig ist. Funktionen, die in einer Bibliothek enthalten sind, können aus einem Sketch, der diese Bibliothek

[22] http://www.wiring.org
[23] http://www.processing.org

einbindet, aufgerufen werden. Der Vorteil dieser Vorgehensweise liegt darin, dass der Umfang von Sketches, die Bibliotheken einbinden, verringert wird und sie dadurch wesentlich leichter zu lesen und zu warten sind.

Zusätzlich können über Bibliotheken die Funktionen des Microcontrollers angesprochen werden, die durch die Arduino-Programmiersprache vor dem Programmierer versteckt werden (vgl. 2.1.2.4). Ein populäres Beispiel dafür ist die Ethernet-Library[24], die zum Lieferumfang der Arduino-IDE gehört. Diese Bibliothek ermöglicht es einem Arduino (der über ein Ethernet-Shield verfügt, vgl. 2.1.2.3), über ein Netzwerkkabel mit einem Server zu kommunizieren. Da für diese Kommunikation *Sockets* (von engl. *„Steckdose“*; Schnittstellen für die Netzwerkkommunikation) benötigt werden und die Arduino-Programmiersprache diese nicht kennt, wird diese Funktionalität durch die Ethernet-Library bereitgestellt, indem in einem in C++ geschriebenen Programm die benötigten Sockets erzeugt und über sie mit einem Server kommuniziert wird.

[24] http://www.arduino.cc/en/Reference/Ethernet

3 Lösungsansätze

An das im Verlauf dieser Arbeit entwickelte Toolkit werden verschiedene Anforderungen gestellt. Neben den bereits in Abschnitt 0 genannten Möglichkeiten, die es bieten soll, handelt es sich dabei hauptsächlich um große Flexibilität im Einsatz, die Offenheit des Quellcodes (Open-Source-Software) und die Möglichkeit, es im vollen Umfang kostenfrei einzusetzen.

In diesem Abschnitt wird beschrieben, welche bereits vorhandenen Lösungen es auf dem Markt gibt, warum diese als nicht geeignet erscheinen, die genannten Anforderungen zu erfüllen, und welche Besonderheiten das ORCHID Toolkit gegenüber existierenden Produkten bietet.

3.1 Marktanalyse

Im Folgenden werden Produkte gezeigt, die eine ähnliche Zielsetzung haben wie das im Rahmen dieser Arbeit entwickelte Toolkit und damit Konkurrenzprodukte sind. Dabei wird analysiert, ob diese Produkte alle Anforderungen erfüllen, die an das ORCHID Toolkit gestellt werden.

3.1.1 Allgemeines

Die Analyse eines Konkurrenzprodukts in den folgenden Abschnitten beinhaltet auch jeweils die Prüfung, ob dieses Produkt neben dem Funktionsumfang, den das ORCHID Toolkit bietet, auch die anderen genannten Anforderungen wie Quelloffenheit und Kostenfreiheit erfüllt.

Proprietäre Produkte oder solche, deren Einsatz die Zahlung von Lizenzgebühren erfordert, gelten im Rahmen dieser Analyse als Lösungen, die als nicht geeignet erscheinen, die gestellten Anforderungen zu erfüllen.

3.1.2 EZ Control

Die Produkte der Reihe „EZ Control“[25] dienen zur Heimautomatisierung. Grundsätzlich wird ein Ansatz verfolgt, der dem des ORCHID Toolkits ähnelt: Steuergeräte werden in ein Netzwerk eingebunden und nehmen Daten von Sensoren entgegen oder schalten Aktoren. Die Bedienung erfolgt durch eine Drittsoftware, z.B. über einen Webbrowser oder ein mobiles Endgerät.

Die folgende Abbildung (Quelle: Herstellerwebsite, [13]) zeigt ein solches Steuergerät.

Abbildung 5 - EZ Control Steuergerät "XS1"

Die Produkte aus der Reihe „EZ Control“ werden als Systemprodukte vertrieben, d.h. dass ein Anwender eine Basisstation (wie in der obigen Abbildung zu sehen) erwirbt und über Lizenzen und entsprechende Firmware verschiedene Funktionen freischaltet.

Heimautomatisierung ist ein potentielles Einsatzgebiet des ORCHID Toolkits, deckt aber nur einen kleinen Teil aller Möglichkeiten ab, die es bietet. Steuergeräte der hier beschriebenen Produktreihe

[25] http://www.ezcontrol.de/

„EZ Control“ erlauben es hingegen nicht, dass beliebige Sensoren und Aktoren mit ihnen verbunden werden, da sie nur mit einer ausgewählten Teilmenge kompatibel sind. Die Hard- und Software der Produktreihe ist nicht quelloffen und wird kommerziell vertrieben.

„EZ Control“ erscheint aus den geschilderten Gründen nicht geeignet, die im Rahmen dieser Arbeit gestellten Anforderungen zu erfüllen.

3.1.3 digitalSTROM.org

Die Organisation „digitalSTROM.org“[26] wurde an der Eidgenössischen Technischen Fachhochschule Zürich gegründet und entwickelt und vertreibt eine Technologie, die die in dieser Arbeit gestellten Anforderungen zu einem großen Teil erfüllt.

Zentraler Bestandteil der Technologie ist ein Mikrochip, der direkt in das in Deutschland vorhandene 230V-Stromnetz eingebunden werden kann. Über diesen Chip werden elektrische Geräte in die Lage versetzt, über das Stromnetz miteinander zu kommunizieren (wenn die kommunizierenden Geräte jeweils über einen Chip verfügen) und sich gegenseitig zu steuern.

Die folgende Abbildung (Quelle: Herstellerwebsite, [14]) zeigt einen solchen Mikrochip. Deutlich erkennbar ist dessen geringe Größe im Vergleich zu einer menschlichen Hand.

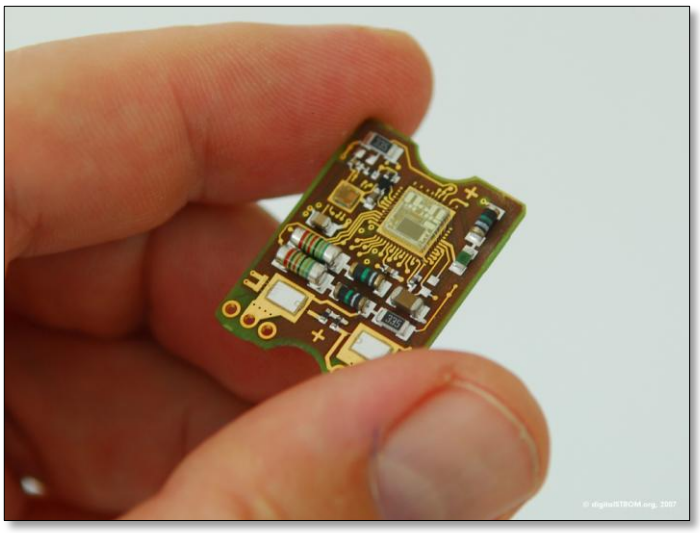

Abbildung 6 - digitalSTROM-Chip

Der gezeigte Chip kann wahlweise direkt vom Hersteller während des Produktionsprozesses in ein elektrisches Gerät eingebaut oder über spezielle Klemmen in ein Kabel integriert werden, über das ein Gerät Strom bezieht.

Spezielle Bauteile ermöglichen es zusätzlich, ein digitalSTROM-Netz an das Internet anzubinden und damit Verbrauchswerte zu ermitteln oder Geräte zu schalten (darüber hinaus kann in Kooperation mit Stromproduzenten ermittelt werden, zu welchem Zeitpunkt der Strom günstig ist und der Verbrauch entsprechend geregelt werden).

Damit deckt die Technologie, die von digitalSTROM.org vertrieben wird, zunächst einen Großteil der Anforderungen ab, die auch das ORCHID Toolkit abdeckt. Der Schwerpunkt in der Ausrichtung dieses Konkurrenzprodukts liegt jedoch eher in der Heimautomatisierung und der Einsparung von Energie, als darin, Hardware mit Software zu verbinden. Sensoren können zwar genutzt werden, wenn

[26] http://www.digitalstrom.org/

sie Teil eines elektrischen Geräts sind, das über einen digitalSTROM-Chip verfügt, können aber nicht wie gefordert direkt mit einem Server verbunden werden. Somit ist beispielsweise die einleitend beschriebene Anbindung einer elektronischen Dartscheibe an einen Server im Netzwerk oder Internet mit dieser Technologie nicht möglich.

Die Technologie ist proprietär und wird nur von wenigen Händlern in Deutschland und der Schweiz vertrieben (die Software-Komponenten sind hingegen unter einer Open-Source-Lizenz veröffentlicht).

Zusammenfassend kann die Technologie, die digitalSTROM.org anbietet, zwar als sehr ähnlich zum ORCHID Toolkit eingestuft werden, erfüllt aber letztlich nicht alle Anforderungen, die im Rahmen dieser Arbeit an eine Werkzeugsammlung zur Integration von Hardware in Softwareprojekte gestellt wurden. Durch das proprietäre Chip-Design, die Tatsache, dass die Technologie nicht kostenfrei zur Verfügung steht und die Ausrichtung auf ein anderes Hauptziel (Energieeinsparung), erscheint die digitalSTROM-Technologie als nicht geeignet, die gestellten Anforderungen zu erfüllen.

3.1.4 Pachube.com

Der Anbieter „Pachube.com“[27] verfolgt eine Strategie, die der in dieser Arbeit vorgestellten sehr ähnlich ist. Über den Internetauftritt von Pachube.com wird ein Webservice angeboten, der es ermöglicht, beliebige Daten über ein auf HTTP basierendes *API (**A**pplication **P**rogramming **I**nterface)* zu senden oder abzurufen.

Um Pachube.com nutzen zu können, muss ein Nutzerkonto angelegt werden. In einer eingeschränkten Basisversion ist die Nutzung kostenlos, die kostenpflichtige Nutzung ermöglicht einen weniger limitierten Zugriff auf den Dienst (eine unlimitierte Nutzung ist aber nicht möglich).

Die folgende Abbildung zeigt das Logo des Anbieters (Quelle: Herstellerwebsite, [15]).

Abbildung 7 - pachube-Logo

Pachube.com setzt, wie auch das ORCHID Toolkit, auf die Datenübermittlung per HTTP. Die Webservice-API kann entsprechend mit den meisten aktuellen Programmiersprachen verwendet werden.

Die von einem Anwender auf den Servern des Anbieters gespeicherten Daten werden in sog. *„Feeds“* aufbereitet und können über das Internet in verschiedenen Formaten abgerufen werden. Diese Feeds sind, mit Ausnahme der teuersten kostenpflichtigen Variante, öffentlich, d.h. jeder kann sie einsehen.

Zunächst erscheint Pachube.com als sehr geeignet, die gestellten Anforderungen zu erfüllen. Diese Einschätzung muss jedoch nach eingehender Prüfung revidiert werden, da der Anbieter trotz aller Ähnlichkeiten einen gänzlich anderen Weg geht als das ORCHID Toolkit. Es gibt zwar die Möglichkeit, den angebotenen Dienst kostenfrei zu nutzen, jedoch ist diese Nutzung stark eingeschränkt[28]: gespeicherte Daten werden nach einem Monat vom Server entfernt, die Webservice-API erlaubt die Unterscheidung von lediglich 5 verschiedenen Datenquellen und es können pro

[27] http://www.pachube.com
[28] https://pachube.com/plans

Minute höchstens 5 Anfragen an den Webservice gestellt werden. Selbst die kostenpflichtige Nutzung ist nicht vollständig unlimitiert.

Der Dienst ermöglicht es, Daten auf dem Server des Anbieters zu speichern und abzurufen, bietet aber keinen Mechanismus zur entfernten Steuerung von Hardware. Ein solcher Mechanismus ist ein zentraler Bestandteil des ORCHID Toolkits.

Der dritte und gravierendste Unterschied zwischen Pachube.com und dem ORCHID Toolkit ist, dass Pachube.com es einem Entwickler weder erlaubt, die serverseitige Verarbeitung und Speicherung von Daten, noch deren Verwendung zu kontrollieren. Auch eine Erweiterung des Diensts durch einen Anwender oder Entwickler ist nicht möglich, da es sich um proprietäre, nicht offene Software handelt. Durch die erzwungene Öffentlichkeit der Feeds (außer in der teuersten kostenpflichtigen Variante) ist darüber hinaus – im Gegensatz zum ORCHID Toolkit – der Zugang zu den gespeicherten Daten für jeden möglich.

Auch wenn Pachube.com einen großen Teil der gestellten Anforderungen abdeckt, erscheint der Dienst durch seine Limitierungen und Kostenpflichtigkeit, den nicht öffentlichen Quellcode, sowie durch die fehlende Kontrolle, die ein Anwender über Daten und deren Verarbeitung hat, als nicht geeignet, die im Rahmen dieser Arbeit gestellten Anforderungen zu erfüllen.

3.1.5 Fazit

Es existieren nur wenige Produkte auf dem Markt, die die Integration einer Hardwareplattform in eine Software ermöglichen. Die wenigen Konkurrenzprodukte, die es gibt, erfüllen stets nur eine Teilmenge der Anforderungen, die im Rahmen dieser Arbeit gestellt werden. Sie sind entweder auf einen anderen Einsatzbereich ausgelegt oder weisen Mängel in Teilbereichen auf. Eine vollständig quelloffene, kostenlose und leicht erweiterbare Lösung ist auf dem Markt nicht vorhanden.

Keines der in diesem Abschnitt geschriebenen Produkte gibt einem Anwender die vollständige Kontrolle über Hard- und Software oder kann alle gestellten Anforderungen erfüllen. Der Bedarf für eine Werkzeugsammlung wie das ORCHID Toolkit kann also als gegeben betrachtet werden.

3.2 Angestrebtes Ergebnis

Im Zuge dieser Arbeit soll ein Toolkit entwickelt und implementiert werden, das einerseits die Probleme und Mängel vorhandener Lösungen nicht wiederholen, andererseits auch neue Wege beschreiten soll. Es soll umfangreiche Funktionen bieten, um Arduino-Clients leicht in eine Software einzubinden, erfahrenen Entwicklern aber auch die Möglichkeit geben, es mit ihren eigenen Klassen zu erweitern und an ihre Bedürfnisse anzupassen (vgl. die Arduino-Philosophie in Abschnitt 2.1.2.4).

3.2.1 Designkriterien

Aufgrund der zahllosen Möglichkeiten, die sich aus der Kombination von Software und Hardware im Zusammenspiel ergeben, ist es unmöglich, alle potentiellen Anwendungsfälle bereits im Vorfeld zu erkennen und in das ORCHID Toolkit zu integrieren.

Vielmehr muss darauf Wert gelegt werden, dass das Toolkit leicht zu erweitern und zu verändern ist. Es wird in allen Phasen der Entwicklung darauf geachtet, dass keine starren Strukturen entstehen, die nicht im Nachhinein verändert und angepasst werden können, und dass alle zentralen Bestandteile des Toolkits flexibel gegen andere austauschbar sind.

Komplexe Vererbungshierarchien werden falls möglich vermieden, um einerseits die Erweiterbarkeit zu erleichtern, andererseits die Wartbarkeit der Software zu erhöhen.

Das ORCHID Toolkit soll leicht zu verwenden sein und es auch weniger erfahrenen Entwicklern ermöglichen, ihre Vorstellungen möglichst schnell umzusetzen. Ein Hauptkriterium beim Entwurf der Software ist entsprechend die Vermeidung hoher Hürden für den Einsatz der Werkzeugsammlung.

3.2.2 Technologiewahl

Durch die starke Auslegung auf Flexibilität und Erweiterbarkeit des Toolkits musste für die Entwicklung eine möglichst zeitgemäße und flexible Plattform gewählt werden (die durch die Arduino-Plattform erzwungenen Limitierungen ließen für die Client-Bibliothek keine freie Wahl zu, weshalb sich das Folgende lediglich auf die Entwicklung der Server-Komponente des Toolkits bezieht).

Aus den positiven Erfahrungen mit C# und dem Microsoft .NET Framework, die ich im Verlauf meiner Bachelorarbeit sammeln konnte (vgl. [16]), fiel die Wahl für diese Arbeit ebenfalls auf C# und, da die Server-Komponente plattformübergreifend ausführbar sein soll, das Mono Framework (das .NET Framework erfordert zur Ausführung Microsoft Windows).

Da sowohl das .NET Framework als auch Mono einen Compiler beinhalten, der Quellcode verschiedener Programmiersprachen in die *CIL* (vgl. 2.3.1) übersetzt, wird eine große Zahl von Programmierern in die Lage versetzt, den Toolkit-Code zu erweitern oder zu verändern, ohne zwingend an eine Programmiersprache gebunden zu sein – andere Plattformen und Programmiersprachen erlauben ein so hohes Maß an Flexibilität nicht. Um die Kompatibilität mit möglichst vielen CIL-Sprachen zu gewährleisten, muss entsprechender Code bestimmte Regeln befolgen. Diese Regeln und deren Umsetzung werden in 5.5.4 und 5.5.5 detailliert beschrieben.

Ein zweiter Aspekt, der zur Auswahl von Mono und C# führte, ist die Sicherheit. Insbesondere Server-Software, die öffentlich über das Internet erreichbar ist, kann schnell zu Sicherheitsproblemen führen, wenn Schwachstellen im Code unerkannt bleiben. Solche Schwachstellen können durch den Einsatz einer modernen Programmiersprache und einer virtuellen Maschine vermieden werden, indem sog. *„managed code“ (deutsch etwa: „verwalteter Code“)* erzeugt wird. Diese Art von Code wird während der Ausführung überwacht. Der Programmierer kann auf die Verwendung von schwachen Datenstrukturen (schwach in Bezug auf deren Sicherheitsaspekte) wie Zeiger verzichten; weiterhin wird Speicher dynamisch verwaltet, so dass *Puffer-Überläufe* (*„Buffer Overflows“*, vgl. [17], S. 7-8) und damit zusammenhängende Schwachstellen nicht auftreten können. Der verwaltete Code ist entsprechend von außen weitaus weniger angreifbar, als es z.B. Code in einer nicht verwalteten Programmiersprache wie C++ ist.

Ein weiterer Vorteil der Wahl von Mono als Entwicklungsplattform für die Server-Komponente des Toolkits ist die Plattformunabhängigkeit. Mono ist für alle verbreiteten Betriebssysteme (Microsoft Windows, Linux und MacOS) verfügbar, die Server-Komponente kann also auf Windows-, Linux- und MacOS-Servern verwendet werden.

Auf die Verwendung von Java[29] als Entwicklungsplattform und Programmiersprache wurde aus Performance- und Effizienzgründen verzichtet. In einer direkten, auf Lasttests basierenden Gegenüberstellung (vgl. [18]) erscheinen Programme, die in Java geschrieben und mit der Java-Laufzeitumgebung ausgeführt werden, zwar zunächst als überlegen in der Geschwindigkeit der Ausführung; es ist jedoch klar zu erkennen, dass in C# geschriebene und mit Mono ausgeführte Programme durchschnittlich nicht nur weniger Hauptspeicher verbrauchen (im Durchschnitt weniger als die Hälfte), sondern auch weniger Code benötigen (im Durchschnitt nur 80% der in Java notwendigen Zeilen), um die Lastest-Berechnungen durchzuführen.

[29] http://www.java.com/en/

Die Verwendung von Mono und C# führt also zu kompakteren Programmen, wodurch die Effizienz sowohl in der Entwicklung, als auch in der Erweiterung und Wartung der Programme erhöht wird. Der Geschwindigkeitsvorteil, den die Verwendung von Java für die Server-Komponente eingebracht hätte, wird durch den zum Teil enorm hohen Speicherbedarf der Java-Programme relativiert. Zusätzlich sind die von C# unter Mono erreichten Geschwindigkeiten ausreichend für den privaten Einsatzbereich, für den das ORCHID Toolkit ausgelegt ist, da dort nicht mit einer großen Anzahl von Clients gerechnet werden muss, die gleichzeitig auf einen ORCHID Server zugreifen oder zeitkritische Operationen durchführen wollen.

4 Anforderungsdefinition

Dieser Abschnitt beschreibt, welche Ziele im Rahmen dieser Arbeit erreicht werden sollen; zudem wird erläutert, welche Anforderungen an die im Zuge dieser Arbeit entwickelte Software gestellt werden und was explizit nicht Bestandteil der Arbeit ist.

4.1 Zielbestimmung

Ziel dieser Arbeit ist die Entwicklung eines Toolkits, auf dessen Grundlage Softwareentwickler in die Lage versetzt werden, nahezu beliebige Hardware (wie bereits einleitend in Abschnitt 0 erwähnt, handelt es sich hier um Hardware, die an einen Arduino Microcontroller angeschlossen werden kann, jegliche andere Hardware wird nicht betrachtet) über einen oder mehrere Arduino Microcontroller in ihre Softwareprojekte einzubinden. Die Einbindung findet dabei über das Internet oder das lokale Netzwerk statt.

Bei der Entwicklung der Bestandteile des Toolkits wird besonders darauf geachtet, dass sie einerseits einfach zu verwenden, andererseits umfangreich und leicht erweiter- und anpassbar sind. Weiterhin wird darauf Wert gelegt, dass das Toolkit als Einheit eine möglichst geringe Kopplung mit der Software eingeht, in die Hardware eingebunden werden soll, damit Änderungen an einem Teil eines Projekts (*„Projekt“* ist hier als Einheit aus einem beliebigen Softwareprojekt und dem im Zuge dieser Arbeit entwickelten Toolkit zu verstehen) den jeweils anderen nicht unmittelbar beeinflussen.

Das ORCHID Toolkit besteht aus zwei Komponenten, die eine Client-Server-Struktur bilden: einer *Server-Komponente* und einer *Client-Bibliothek*. Die Server-Komponente verfügt über zwei Schnittstellen (*„Server“*), die in der Lage sind, spezielle HTTP-Anfragen von Clients und Drittsoftware entgegen zu nehmen. Ein Client ist dabei ein Arduino Microcontroller, in dessen Programm die Client-Bibliothek eingebunden wurde; Drittsoftware ist jede Software, die über das Toolkit in die Lage versetzt werden soll, mit Hardware zu kommunizieren.

Die Kommunikation zwischen ORCHID Server und ORCHID Client bzw. ORCHID Server und Drittsoftware ist bidirektional, d.h. dass Daten sowohl an die Server-Komponente (z.B. Sensordaten durch einen Client oder Instruktionen für einen Client durch eine Drittsoftware), als auch von ihr (z.B. gesammelte Sensordaten eines Clients oder Instruktionen für einen Client) gesendet werden können. Der Ursprung der Kommunikation ist dabei immer ein Client oder eine Drittsoftware, niemals der ORCHID Server selbst; die Server-Komponente folgt damit der Arbeitsweise regulärer Webserver wie z.B. der des Apache HTTP Servers der Apache Software Foundation[30].

Über die zwei Schnittstellen der Server-Komponente werden Clients und Drittsoftware in die Lage versetzt, miteinander kommunizieren und aufeinander reagieren zu können.

4.1.1 Musskriterien

Die im Folgenden beschriebenen Produktfunktionen verstehen sich als Musskriterien, sind also zwingend Teil des Funktionsumfangs des Toolkits. Sie sind aufgeteilt in Musskriterien für die Server-Komponente und Musskriterien für die Client-Bibliothek. Zusätzlich werden Musskriterien für eine beispielhafte Drittsoftware genannt, die nicht direkt Bestandteil dieser Arbeit ist, sondern als Machbarkeitsnachweis (*„Proof of Concept“*) entwickelt wird.

4.1.1.1 Server-Komponente

Das Toolkit muss es einem Softwareentwickler ermöglichen, möglichst einfach einen voll funktionsfähigen Server im lokalen Netzwerk oder Internet aufsetzen zu können, der als Vermittler in der Kommunikation zwischen Clients und einer Software auftritt. Zu diesem Zweck muss die Server-

[30] http://httpd.apache.org/

Komponente zahlreiche Funktionen anbieten, die an dieser Stelle stichpunktartig genannt und im weiteren Verlauf der Arbeit detailliert erläutert werden.

- Warten auf Verbindungen durch Clients
- Warten auf Verbindungen durch Drittsoftware
- Akzeptieren von Client-Verbindungen
- Akzeptieren von Drittsoftware-Verbindungen
- Auslesen und Transformieren (*„Parsen“*) von HTTP-Anfragen (HTTP-Requests) von Clients
- Auslesen und Transformieren (*„Parsen“*) von HTTP-Anfragen (HTTP-Requests) von Drittsoftware
- Erkennen und Unterscheiden von verschiedenen HTTP-Requests (d.h. HTTP-Requests, die eine bestimmte *HTTP-Methode* verwenden, vgl. 2.2)
- Verifizieren der Gültigkeit eingehender HTTP-Anfragen
- Speichern von durch Clients übertragenen Sensordaten in einer Datenbank
- Abrufen von Sensordaten aus einer Datenbank
- Speichern von durch Drittsoftware übertragenen Instruktionen für Clients in einer Datenbank
- Abrufen von Instruktionen aus einer Datenbank
- Abrufen von Informationen über Clients, Sensoren und Aktoren aus einer Datenbank
- Senden von Sensordaten als HTTP-Antwort an Drittsoftware
- Senden von Instruktionen als HTTP-Antwort an Clients
- Senden von Informationen über Clients, Sensoren und Aktoren als HTTP-Antwort an Drittsoftware
- Senden von Fehlermeldungen als HTTP-Antwort an Clients
- Senden von Fehlermeldungen als HTTP-Antwort an Drittsoftware

Zusätzliches Musskriterium ist die exemplarische Umsetzung einer umfassend konfigurierbaren Referenzimplementierung eines ORCHID Servers, d.h. einer Software, die die Server-Komponente verwendet, um mit ORCHID Clients und Drittsoftware zu kommunizieren, bzw. als Vermittler in der Kommunikation zwischen diesen dient.

4.1.1.2 Client-Bibliothek

Die Client-Bibliothek dient dazu, Entwickler in die Lage zu versetzen, einen oder mehrere Clients mit der Server-Komponente kommunizieren zu lassen. Um diese Aufgabe zu bewältigen, muss die Bibliothek die folgenden Funktionen bereitstellen, die hier lediglich genannt, im weiteren Verlauf der Arbeit aber detailliert beschrieben werden:

- Aufbau einer HTTP-Verbindung zu einem ORCHID Server
- Senden von gemessenen Sensordaten an einen ORCHID Server
- Senden von Heartbeats[31] an einen ORCHID Server
- Anfordern von Anweisungen von einem ORCHID Server
- Auslesen von Anweisungen, die als HTTP-Antwort von einem ORCHID Server gesendet wurden
- Ausführen einer von einem ORCHID Server erhaltenen Anweisung

Zusätzliches Musskriterium ist die Implementierung eines Referenz-Sketchs, der die o.g. Funktionen verwendet, um mit einem ORCHID Server zu kommunizieren.

[31] Eine *„Heartbeat-Anfrage“* (ein *„Heartbeat“*) ist eine Anfrage, die dem Server lediglich mitteilt, dass der sendende Client bereit ist, vgl. auch 5.8.

4.1.1.3 Drittsoftware

Da das ORCHID Toolkit einen Entwickler in die Lage versetzen soll, beliebige Hardware (vgl. Abschnitt 0) in eigene Softwareprojekte einzubinden, muss dieser Arbeit als Machbarkeitsnachweis eine entsprechende Software beiliegen, die die in diesem Abschnitt beschriebenen Funktionen des Toolkits nutzt, um mit Hardware zu kommunizieren. Aus diesem Grund wird als dritter Softwarebestandteil der Arbeit eine PHP-Bibliothek entwickelt, die die folgenden Kriterien erfüllt:

- Abrufen von Sensordaten von einem ORCHID Server (einzelner Datensatz)
- Abrufen von Sensordaten von einem ORCHID Server (mehrere Datensätze)
- Hochladen von Anweisungen für einen ORCHID Client auf einen ORCHID Server
- Abrufen von Informationen über einen ORCHID Client von einem ORCHID Server

Eine – ebenfalls mit PHP entwickelte – Website demonstriert mittels dieser Bibliothek die Funktionsweise des Toolkits und dient somit als Proof of Concept. Die hier beschriebene Drittsoftware ist kein Bestandteil des eigentlichen Toolkits, sondern lediglich ein Machbarkeitsnachweis. Ihre Anforderungen werden an dieser Stelle nur aus Gründen der Vollständigkeit genannt.

4.1.2 Wunschkriterien

Die folgenden Kriterien sind nicht explizit Bestandteil dieser Arbeit und verstehen sich als optionale Wunschkriterien; ihre Implementierung ist nicht verpflichtender Bestandteil des Toolkits, die Möglichkeit einer späteren Implementierung ist aber durch den Aufbau des Toolkits gegeben.

4.1.2.1 Server-Komponente

- Versenden von Nachrichten zwischen zwei ORCHID Clients (über einen ORCHID Server)
- Verwaltung von Berechtigungen für ORCHID Clients
- Speichern von durch einen ORCHID Client übertragenen Instruktionen für einen anderen ORCHID Client
- Sicherheitsfunktionen und Zugriffsregulierungen

4.1.2.2 Client-Bibliothek

- Hochladen von Nachrichten für einen ORCHID Client auf einen ORCHID Server
- Abrufen von Nachrichten von einem ORCHID Server
- Hochladen von Instruktionen für einen ORCHID Client auf einen ORCHID Server

4.1.2.3 Drittsoftware

- Senden von Nachrichten für einen ORCHID Client an einen ORCHID Server

4.1.3 Abgrenzungskriterien

Die im Folgenden genannten Kriterien sind ausdrücklich nicht Bestandteil dieser Arbeit. Ihre Implementierung in das Toolkit ist nicht vorgesehen (aber nicht unmöglich).

- Vollständige Kompatibilität zu HTTP/1.1
- Verschlüsselung der Kommunikation zwischen Kommunikationspartnern
- Nutzungsmöglichkeit als Webserver oder Browser

Bibliotheken, die von Entwicklern in ihren Softwareprojekten genutzt werden können, um mit der Server-Komponente zu kommunizieren, sind nicht Bestandteil der Arbeit. Aufgrund der hohen Flexibilität des Toolkits und dessen Unabhängigkeit von der Implementierung von Drittsoftware ergeben sich so viele verschiedene Möglichkeiten, dass es unmöglich ist, diese im Rahmen einer Arbeit wie dieser abzudecken. Als einzige Ausnahme wird die in 4.1.1.3 beschriebene Drittsoftware als Proof of Concept entwickelt. Weitere Bibliotheken sind nicht Teil dieser Arbeit.

4.2 Produkteinsatz

Das ORCHID Toolkit wird unter der Prämisse entwickelt, dass es von möglichst vielen Entwicklern in deren Softwareprojekten eingesetzt werden kann. Aufgrund der Vielfalt von Möglichkeiten, die sich aus der Kombination von Arduino Microcontrollern und Drittsoftware ergibt, ist es schwierig, eine einzelne Definition für den Einsatzbereich des Toolkits zu finden.

Die Möglichkeiten für den Einsatz des ORCHID Toolkits ergeben sich stattdessen aus den vorgesehenen Anwendungsbereichen, der Zielgruppe und der Lizenz, unter der es veröffentlicht wird. Abschließend werden einige Beispiele für den Einsatz des Toolkits gegeben.

4.2.1 Anwendungsbereich

Das ORCHID Toolkit ist auf den gleichen Anwendungsbereich ausgelegt, wie auch die Arduino Microcontroller-Plattform, mit der hauptsächlich Prototypen und nur selten Projekte für Produktivumgebungen entwickelt werden.

Entsprechend sollen auch mit dem Toolkit zum größten Teil Prototypen für Softwareprojekte entwickelt werden; ein Einsatz in einer Produktivumgebung ist möglich, aber nicht vorgesehen.

Das Toolkit ist explizit nicht für den Einsatz in sicherheitskritischen Umgebungen ausgelegt (Details zu dieser Einschränkung können 6.6.1 entnommen werden).

4.2.2 Zielgruppe

Zielgruppe des Toolkits sind private Softwareentwickler, die über einen oder mehrere Arduino Microcontroller angeschlossene Hardware in ein Softwareprojekt einbinden wollen und mindestens über Grundkenntnisse in der Server-Administration, der Softwareentwicklung und der Entwicklung mit dem Arduino Microcontroller verfügen. Zum Anschluss von Hardware an den Arduino sind natürlich ebenfalls entsprechende Kenntnisse notwendig.

Als wichtige Einschränkung der Zielgruppe gilt, dass das Toolkit nicht für kommerzielle Projekte eingesetzt werden darf (vgl. 4.2.3). Firmen gehören nicht per Definition zur Zielgruppe, können das Toolkit aber zur Entwicklung eigener Prototypen verwenden, sofern diese nicht kommerziell veröffentlicht werden.

4.2.3 Lizenzbedingungen

Das ORCHID Toolkit wird unter der Lizenz *"Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported"* veröffentlicht, deren Lizenztext unter [19] in englischer und deutscher Sprache verfügbar ist. Aufgrund des langen Namens der Lizenz wird sie häufig als „*CC BY-NC-SA 3.0*" bezeichnet. Das Logo dieser Lizenz wird in Abbildung 8 [32] gezeigt.

Abbildung 8 - Logo der CC BY-NC-SA 3.0

Im Detail erlaubt es die CC BY-NC-SA 3.0 einer Person, ein Werk zu kopieren und zu verteilen, sowie es zu verändern und auch eigene Projekte daraus abzuleiten, solange diese Person den eigentlichen Autor benennt (*Attribution*), das Werk nicht für kommerzielle Zwecke verwendet (*Noncommercial*) und jegliche veröffentlichte Verwendung des Werks (inkl. Veränderungen,

[32] Quelle: http://mirrors.creativecommons.org/presskit/buttons/88x31/png/by-nc-sa.eu.png

Erweiterungen und darauf aufbauender Werke) unter der gleichen oder einer ähnlichen, voll kompatiblen Lizenz veröffentlicht (*ShareAlike*).

Ein weiterer großer Vorteil dieser Lizenz ist die Möglichkeit, dass der eigentliche Autor eines Werks (der Inhaber des Urheberrechts) jede der o.g. Bedingungen einzeln für eine einzelne Person (oder Gruppe von Personen) aufheben kann, wenn er dies möchte.

Eine komfortable Übersicht über die Lizenz kann unter [20] in englischer Sprache eingesehen werden.

4.2.4 Einsatzmöglichkeiten

Die potentiellen Einsatzmöglichkeiten für das ORCHID Toolkit sind sehr zahlreich und hängen stets von den Vorstellungen eines Entwicklers ab. Entsprechend sind die im Folgenden beschriebenen Möglichkeiten nicht als vollständige Liste zu verstehen.

4.2.4.1 Heimautomatisierung

Auf Basis des Toolkits können Lösungen zur Heimautomatisierung entwickelt werden. Das bedeutet, dass über eine zentrale Schnittstelle (z.B. einen Server im Internet oder Netzwerk), Haushaltsgeräte angesteuert und geschaltet werden können.

Denkbare Szenarien für eine solche Automatisierung sind u.a. die programm- oder zeitgesteuerte Schaltung von Lampen in einer Wohnung oder einem Haus aus der Ferne, die Überwachung bestimmter Haushaltsgeräte, deren unbeaufsichtigter Betrieb eine Gefahr darstellen könnte (z.B. Bügeleisen oder Kaffeemaschinen) oder auch die Einsparung von Strom über die Schaltung von Geräten abhängig von gemessenen Sensorwerten (hier insbesondere Heizung und Licht).

4.2.4.2 Überwachung

Da an einen Arduino Microcontroller zahlreiche Sensoren angeschlossen werden können, die unterschiedlichste Umgebungswerte wie Luftfeuchtigkeit, Temperatur, Menge des einfallenden Lichts, Umgebungslautstärke oder auch die Konzentration bestimmter Gase in der Luft messen, kann auf Basis des Toolkits ein umfassendes System zur Überwachung eines Gebäudes oder eines einzelnen Raums realisiert werden.

Ist der ORCHID Server dabei über das Internet erreichbar, kann der überwachte Ort zu jeder Zeit von jedem Ort der Welt aus überprüft werden. In Kombination mit den im vorherigen Abschnitt beschriebenen Möglichkeiten zur Automatisierung können erkannte problematische Zustände ebenfalls aus der Ferne behoben werden.

Ein Beispiel für diesen Anwendungsfall ist die Überwachung eines Gewächshauses. Entsprechende Sensoren messen in regelmäßigen Abständen die relevanten Werte wie Luftfeuchtigkeit und Temperatur und leiten diese an den Server weiter. Von diesem kann eine entsprechende Drittsoftware die gemessenen Daten abrufen und der überwachenden Person aufbereitet darstellen. Weichen die ermittelten Werte von den gewünschten ab, können über Steuerelemente in der Drittsoftware entsprechende Aktionen ausgelöst werden, um diese Diskrepanz zu beheben (z.B. das Erhöhen der Umgebungstemperatur).

4.2.4.3 Aufgabenautomatisierung

Die ursprüngliche Idee zu dieser Arbeit entstand weder durch das Bedürfnis nach Heimautomatisierung, noch durch das nach Überwachung von bestimmten Zuständen aus der Ferne, sondern vielmehr durch den Wunsch, eine Aufgabe (Digitalisierung von Spielergebnissen, vgl. Abschnitt 0) zu automatisieren.

Die Idee, eine elektronische Dartscheibe so zu modifizieren, dass erzielte Treffer ohne weitere Aufwände an einen Server übermittelt und von einer Drittsoftware grafisch und statistisch aufbereitet werden können, ist dabei nur eine der zahlreichen Möglichkeiten zur Automatisierung einer solchen Aufgabe. Viele elektronische Geräte können – oft auch ohne Modifikation der Geräte selbst – über das Toolkit an einen ORCHID Server angebunden und über diesen von einer Drittsoftware angesteuert werden.

4.2.4.4 Künstlerische Projekte

Ebenso wie die Automatisierung bestimmter Aufgaben oder ganzer Wohnbereiche, sind mit dem ORCHID Toolkit auch künstlerische Projekte realisierbar („künstlerisch“ bezieht sich hier auf die kreative Verwendung elektronischer Bauteile).

Im September 2001 begann ein Projekt des Chaos Computer Club[33] (CCC) in Berlin namens *„Blinkenlights“*[34]. Dabei wurden die oberen acht Stockwerke eines Gebäudes (*„Haus des Lehrers“* am Alexanderplatz in Berlin) mit Lampen versehen, die von einem zentralen Computer gesteuert werden konnten. Das Projekt bot u.a. die Möglichkeit, Pixelgrafiken (maximal 8x18 Pixel) mit diesen Lampen darzustellen.

Die folgende Abbildung 9 [35] (aufgenommen durch Tim Pritlove[36], einen der Mitbegründer des Projekts) zeigt eine Luftansicht von *„Blinkenlights“* vom Berliner Fernsehturm aus. Im mittleren Bereich ist das Haus des Lehrers zu sehen, auf dessen Fassade zum Zeitpunkt der Aufnahme ein Herz dargestellt wurde.

Abbildung 9 - Blinkenlights

Eine solche Installation kann – entsprechenden Aufwand vorausgesetzt – auch mit dem ORCHID Toolkit realisiert werden.

[33] http://www.ccc.de
[34] http://www.blinkenlights.net
[35] Quelle: http://commons.wikimedia.org/wiki/File:Project-blinkenlights-aerial-view.jpg
[36] http://tim.pritlove.org/

4.3 Produktanforderungen

Im Folgenden werden die Anforderungen an Hard- und Software beschrieben, die das ORCHID Toolkit an ein ausführendes System stellt. Diese gliedern sich in Anforderungen an die Software eines Systems, also u.a. an das verwendete Betriebssystem, und in Anforderungen an die Hardware, also beispielweise die Geschwindigkeit des Prozessors oder die Größe des verfügbaren Arbeitsspeichers.

4.3.1 Softwareanforderungen

Die Softwareanforderungen für das ORCHID Toolkit teilen sich auf in Anforderungen für die Ausführung der Server-Komponente und Anforderungen für die Verwendung der Client-Bibliothek.

4.3.1.1 Anforderungen der Server-Komponente

Die Server-Komponente kann plattformübergreifend unter Microsoft Windows (empfohlen: Windows 7, 32 Bit), Linux (empfohlen: Ubuntu 10.10 *„Maverick Meerkat“*, 64 Bit) und MacOS (empfohlen: MacOS X 10.6, *„Snow Leopard“*) verwendet werden. Voraussetzung für die Ausführung des Servers ist ein installiertes Mono Framework (Version 2.6.7 oder kompatibel) sowie eine vom ausführenden Server erreichbare MySQL-Instanz (Version 5.5.8 oder kompatibel).

Zur Erweiterung des Servers um eigene Klassen ist ebenfalls ein installiertes Mono Framework (Version 2.6.7 oder kompatibel) und optional eine Entwicklungsumgebung (MonoDevelop Version 2.4.2.1 oder kompatibel) notwendig.

4.3.1.2 Anforderungen der Client-Bibliothek

Auch die Client-Bibliothek kann plattformübergreifend unter Microsoft Windows, Linux und MacOS verwendet werden. Einzige Voraussetzung für die Verwendung ist eine installierte Arduino-IDE (Version 0018 oder kompatibel, benötigt Java).

4.3.2 Hardwareanforderungen

Server-Komponente und Client-Bibliothek stellen unterschiedliche Anforderungen an die zur Ausführung benötigte Hardware, die im Folgenden erläutert werden. Die hier geschilderten Anforderungen wurden unter der Prämisse ermittelt, dass sowohl die Server-Komponente, als auch die Client-Bibliothek flüssig und stabil ausgeführt werden können.

4.3.2.1 Anforderungen der Server-Komponente

Um die Server-Komponente ausführen zu können, muss ein entsprechendes System über einen Zugang zum Netzwerk bzw. zum Internet verfügen. Ein schneller Mehrkernprozessor erhöht die Ausführungsgeschwindigkeit, die Server-Komponente kann aber auch auf Systemen mit Einzelkernprozessor ausgeführt werden.

Nahezu jedes Computersystem, auf dem ein zur Ausführung notwendiges Betriebssystem (wie in 4.3.1.1 beschrieben) lauffähig ist, kann auch die Server-Komponente problemlos ausführen.

Als Mindestanforderung für die Verwendung der Server-Komponente gelten:

- **Prozessor** 1 GHz Pentium (oder kompatibel)
- **Arbeitsspeicher** 1 Gigabyte
- **Festplattenspeicher** 1 Gigabyte freier Speicherplatz

Empfohlen wird eine Umgebung, die mindestens die folgenden Leistungsmerkmale aufweist:

- **Prozessor** 1,8 GHz Dual-Core Pentium (oder kompatibel)
- **Arbeitsspeicher** 2 Gigabyte
- **Festplattenspeicher** 4 Gigabyte freier Speicherplatz

Die hohen Anforderungen an den freien Festplattenspeicherplatz ergeben sich aus den ggf. anfallenden großen Datenmengen, die durch die Clients verursacht werden, wenn sie Sensordaten an den Server übermitteln.

4.3.2.2 Anforderungen der Client-Bibliothek

Die Client-Bibliothek ist für Arduino Microcontroller-Boards ausgelegt, erfordert also die Verwendung eines Arduino I/O-Boards. Andere Microcontroller-Boards werden nicht unterstützt.

Die Mindestanforderungen an ein verwendetes Arduino I/O-Board entsprechen der Spezifikation des Arduino *Duemilanove* und lauten wie folgt:

- **Microcontroller** Atmel ATmega168
- **Flash-Speicher** 32 Kilobyte
- **Arbeitsspeicher** 2 Kilobyte

Desweiteren wird zur Verwendung der Client-Bibliothek ein Ethernet Shield benötigt[37].

Insbesondere der verfügbare Arbeits- und Flash-Speicher des verwendeten I/O-Boards ist für die stabile Ausführung der Client-Bibliothek essentiell (vgl. 6.6.4).

4.3.3 Produktschnittstellen

Die einzige direkte Produktschnittstelle zu anderen Systemen ist die Anbindung der Server-Komponente an eine MySQL-Datenbank. Alle anderen Schnittstellen basieren auf HTTP und werden gesondert in 5.8 beschrieben.

Die Client-Bibliothek bietet keine Schnittstellen zu anderen Systemen.

4.4 Produktfunktionen

Im Folgenden werden die Produktfunktionen des Toolkits beschrieben, die sich aus den in 4.1 beschriebenen Muss- und Wunschkriterien ergeben. Jede Funktion ist dazu in einer eigenen Tabelle untergebracht, in der sie kurz beschrieben wird. Zusätzlich wird genannt, welche Bedingungen an vorhergehende Ereignisse oder Voraussetzungen die Funktion hat, was die Konsequenzen für den Erfolgs- oder Fehlerfall sind, und, falls notwendig, welche Datenstrukturen benötigt werden. Wenn es für das Verständnis der beschriebenen Funktion von Vorteil ist, werden zudem ausgesuchte beteiligte Klassen genannt.

Für alle hier genannten Produktfunktionen, an denen ein ORCHID Client als anfragender Verbindungspartner beteiligt ist, gilt, dass die entsprechende Funktion nur dann erfolgreich ist, wenn der Client auch die Berechtigung hat, die auslösende Aktion durchzuführen. Da Berechtigungen jedoch ein Wunschkriterium sind (vgl. 4.1.2), wird diese Bedingung nicht in den Vorbedingungen der entsprechenden Produktfunktionen erwähnt. Eine Anfrage eines ORCHID Clients, der nicht berechtigt ist, die entsprechende Produktfunktion zu nutzen, wird – bei Umsetzung der Berechtigungsverwaltung und entsprechender Server-Konfiguration – stets mit einer Fehlermeldung (HTTP-Code 401, *„Unauthorized“*) beantwortet.

[37] Seit dem 13. Juli 2011 ist das „Arduino Ethernet“-I/O-Board erhältlich. Es verfügt über einen integrierten Ethernet-Port und kann entsprechend ohne Ethernet Shield betrieben werden. Informationen unter http://arduino.cc/en/Main/ArduinoBoardEthernet

4.4.1 Funktionen der Server-Komponente

/FS10/	Serverkonfiguration einlesen
Vorbedingung	Konfigurationsdatei vorhanden und lesbar
Erfolgsfall	Serverweite Konfiguration verfügbar
Fehlerfall	Server-Instanzen können nicht gestartet werden, Abbruch und Neustart der Server-Komponente
Beschreibung	Die serverweite, zentrale Konfigurationsdatei wird eingelesen
Beteiligte	`Orchid.Core.Configuration.Config`
Kategorie	Musskriterium

Tabelle 5 - FS10: Serverkonfiguration einlesen

/FS20/	Hardware-Server starten
Vorbedingung	/FS10/
Erfolgsfall	Hardware-Server wartet auf eingehende Verbindungen an dem ihm zugewiesenen Port
Fehlerfall	Abbruch und Neustart der Server-Komponente
Beschreibung	Die Instanz des Hardware-Servers wird gestartet
Beteiligte	`Orchid.Core.Servers.OrchidHardwareServer`
Kategorie	Musskriterium

Tabelle 6 - FS20: Hardware-Server starten

/FS30/	Software-Server starten
Vorbedingung	/FS20/
Erfolgsfall	Software-Server wartet auf eingehende Verbindungen an dem ihm zugewiesenen Port
Fehlerfall	Abbruch und Neustart der Server-Komponente
Beschreibung	Die Instanz des Software-Servers wird gestartet
Beteiligte	`Orchid.Core.Servers.OrchidSoftwareServer`
Kategorie	Musskriterium

Tabelle 7 - FS30: Software-Server starten

/FS40/	Auf eingehende Client-Verbindung warten
Vorbedingung	/FS20/
Erfolgsfall	Es wird auf eingehende Verbindungen eines ORCHID Client gewartet
Fehlerfall	Hardware-Server kann nicht auf eingehende Verbindungen warten, Abbruch und Neustart der Server-Komponente
Beschreibung	Der Hardware-Server wartet an dem ihm zugewiesenen Port auf eingehende Verbindungen von ORCHID Clients
Beteiligte	`Orchid.Core.Servers.OrchidHardwareServer`
Kategorie	Musskriterium

Tabelle 8 - FS40: Auf eingehende Client-Verbindung warten

/FS50/	Auf eingehende Drittsoftware-Verbindung warten
Vorbedingung	/FS30/
Erfolgsfall	Es wird auf eingehende Verbindungen einer Drittsoftware gewartet
Fehlerfall	Software-Server kann nicht auf eingehende Verbindungen warten, Abbruch und Neustart der Server-Komponente
Beschreibung	Der Software-Server wartet an dem ihm zugewiesenen Port auf eingehende Verbindungen von Drittsoftware
Beteiligte	`Orchid.Core.Servers.OrchidSoftwareServer`
Kategorie	Musskriterium

Tabelle 9 - FS50: Auf eingehende Drittsoftware-Verbindung warten

/FS60/	Eingehende Client-Verbindung annehmen
Vorbedingung	/FS40/
Erfolgsfall	Eine eingehende Verbindung eines ORCHID Client wird angenommen, die Verarbeitung der Anfrage beginnt
Fehlerfall	Hardware-Server kann eingehende Verbindung nicht annehmen, die Anfrage kann nicht verarbeitet werden
Beschreibung	Geht eine Verbindung an dem Port ein, an dem der Hardware-Server auf eingehende Verbindungen wartet, nimmt er diese entgegen und leitet sie an eine neu erzeugte Instanz eines Hardware-Daemons weiter
Beteiligte	`Orchid.Core.Servers.OrchidHardwareServer`, `Orchid.Core.Daemons.OrchidHardwareDaemon`
Kategorie	Musskriterium

Tabelle 10 - FS60: Eingehende Client-Verbindung annehmen

/FS70/	Eingehende Drittsoftware-Verbindung annehmen
Vorbedingung	/FS50/
Erfolgsfall	Eine eingehende Verbindung einer Drittsoftware wird angenommen, die Bearbeitung der Anfrage beginnt
Fehlerfall	Software-Server kann eingehende Verbindung nicht annehmen, die Anfrage kann nicht bearbeitet werden
Beschreibung	Geht eine Verbindung an dem Port ein, an dem der Software-Server auf eingehende Verbindungen wartet, nimmt er diese entgegen und leitet sie an eine neu erzeugte Instanz eines Software-Daemons weiter
Beteiligte	`Orchid.Core.Servers.OrchidSoftwareServer`, `Orchid.Core.Daemons.OrchidSoftwareDaemon`
Kategorie	Musskriterium

Tabelle 11 - FS70: Eingehende Drittsoftware-Verbindung annehmen

/FS80/	Eingehende Anfrage auslesen
Vorbedingung	/FS60/ bzw. /FS70/
Erfolgsfall	Die eingehende HTTP-Anfrage wurde vollständig ausgelesen und kann verwendet werden
Fehlerfall	Die eingehende HTTP-Anfrage kann nicht verarbeitet werden
Beschreibung	Eine HTTP-Anfrage liegt als Stream (`BufferedStream`) vor. Dieser wird byteweise ausgelesen und die gelesenen Inhalte zwischengespeichert.
Datenstrukturen	`System.IO.BufferedStream`
Beteiligte	`Orchid.Core.Parsers.OrchidHttpRequestParser`
Kategorie	Musskriterium

Tabelle 12 - FS80: Eingehende Anfrage auslesen

/FS90/	Gelesene Anfrage transformieren
Vorbedingung	/FS80/
Erfolgsfall	Die HTTP-Anfrage wurde in eine spezielle Datenstruktur überführt
Fehlerfall	Die Anfrage kann nicht verarbeitet werden
Beschreibung	Nach dem Auslesen einer HTTP-Anfrage liegen die darin enthaltenen Daten als Text vor. Sie werden zur leichteren und sichereren Verwendung in eine spezielle Datenstruktur (`OrchidHttpRequest`) überführt.
Datenstrukturen	`Orchid.Core.Requests.OrchidHttpRequest`
Beteiligte	`Orchid.Core.Parsers.OrchidHttpRequestParser`
Kategorie	Musskriterium

Tabelle 13 - FS90: Gelesene Anfrage transformieren

/FS100/	Request-Typ bestimmen
Vorbedingung	/FS90/
Erfolgsfall	Der Typ der eingehenden Anfrage wurde bestimmt
Fehlerfall	Die Anfrage kann nicht verarbeitet werden, Fehlermeldung an Verbindungspartner (ORCHID Client oder Drittsoftware)
Beschreibung	Der Typ eines Requests wird über die verwendete HTTP-Methode (vgl. 2.2) bestimmt. Nur wenn die Methode bestimmt werden kann und bekannt ist, kann eine Anfrage verarbeitet werden, ansonsten wird eine Fehlermeldung übermittelt.
Datenstrukturen	`Orchid.Core.Requests.OrchidHttpRequest`, `Orchid.Core.HttpRequestType`
Beteiligte	`Orchid.Core.Daemons.OrchidHardwareDaemon` oder `Orchid.Core.Daemons.OrchidSoftwareDaemon`
Kategorie	Musskriterium

Tabelle 14 - FS100: Request-Typ bestimmen

/FS110/	Sensordaten speichern
Vorbedingung	/FS100/, Request-Typ *„PUT“*, angenommen durch Hardware-Server
Erfolgsfall	Sensordaten wurden gespeichert, Erfolgsmeldung an Client übermittelt
Fehlerfall	Sensordaten können nicht gespeichert werden, Fehlermeldung an Client
Beschreibung	Sendet ein Client einen PUT-Request, wird dieser entgegengenommen und von einer Instanz von `OrchidHardwareDaemon` verarbeitet. Die übermittelten Sensordaten werden ausgelesen und gespeichert.
Datenstrukturen	`Orchid.Core.Requests.OrchidHttpRequest`, `Orchid.Core.HttpRequestType`, `Orchid.Core.Data.SensorData`
Beteiligte	`Orchid.Core.Daemons.OrchidHardwareDaemon`, `Orchid.Core.Data.MySqlAdapter`
Kategorie	Musskriterium

Tabelle 15 - FS110: Sensordaten speichern

/FS120/	Sensordaten abrufen
Vorbedingung	/FS100/, Request-Typ „GET“, angenommen durch Software-Server
Erfolgsfall	Sensordaten wurden abgerufen und an Drittsoftware übermittelt
Fehlerfall	Sensordaten können nicht abgerufen werden, Fehlermeldung an Drittsoftware
Beschreibung	Ein GET-Request, der von einer Drittsoftware übermittelt wurde, wird von einer Instanz von `OrchidSoftwareDaemon` verarbeitet. Aus dem Request wird ausgelesen, welche Sensordaten (und in welcher Menge) die Drittsoftware erhalten möchte, diese Daten aus der Datenbank ausgelesen und an die Drittsoftware übermittelt.
Datenstrukturen	`Orchid.Core.Requests.OrchidHttpRequest`, `Orchid.Core.HttpRequestType`, `Orchid.Core.Data.SensorData`
Beteiligte	`Orchid.Core.Daemons.OrchidSoftwareDaemon`, `Orchid.Core.Data.MySqlAdapter`
Kategorie	Musskriterium

Tabelle 16 - FS120: Sensordaten abrufen

/FS130/	Heartbeat-Request verarbeiten
Vorbedingung	/FS100/, Request-Typ „TRACE“, angenommen durch Hardware-Server
Erfolgsfall	Heartbeat wurde verarbeitet, Datensatz in der Datenbank aktualisiert, Erfolgsmeldung an Client gesendet
Fehlerfall	Datensatz kann nicht aktualisiert werden, Fehlermeldung an Client
Beschreibung	Ein TRACE-Request eines Clients wird von einer Instanz von `OrchidHardwareDaemon` verarbeitet. Die im Request enthaltenen Daten werden gelesen und der Client in der Datenbank aktualisiert.
Datenstrukturen	`Orchid.Core.Requests.OrchidHttpRequest`, `Orchid.Core.HttpRequestType`
Beteiligte	`Orchid.Core.Daemons.OrchidHardwareDaemon`, `Orchid.Core.Data.MySqlAdapter`
Kategorie	Musskriterium

Tabelle 17 - FS130: Heartbeat-Request verarbeiten

/FS140/	Client-Instruktion abrufen
Vorbedingung	/FS100/, Request-Typ „GET“, angenommen durch Hardware-Server
Erfolgsfall	Instruktion für Client wurde aus der Datenbank ausgelesen, in der Datenbank aktualisiert und an den anfragenden Client übermittelt
Fehlerfall	Datensatz kann nicht aktualisiert werden; Fehlermeldung an Client
Beschreibung	Ein GET-Request eines Clients wird von einer Instanz von *OrchidHardwareDaemon* verarbeitet. Die gesuchte Instruktion wird aus der Datenbank abgerufen und an den anfragenden Client übermittelt.
Datenstrukturen	*Orchid.Core.Requests.OrchidHttpRequest, Orchid.Core.HttpRequestType, Orchid.Core.Data.Instruction*
Beteiligte	*Orchid.Core.Daemons.OrchidHardwareDaemon, Orchid.Core.Data.MySqlAdapter*
Kategorie	Musskriterium

Tabelle 18 - FS140: Client-Instruktion abrufen

/FS150/	Client-Instruktion speichern
Vorbedingung	/FS100/, Request-Typ „PUT“, angenommen durch Software-Server
Erfolgsfall	Instruktion für einen Client wurde in der Datenbank gespeichert, Erfolgsmeldung an Drittsoftware gesendet
Fehlerfall	Instruktion kann nicht gespeichert werden, Fehlermeldung an Software
Beschreibung	Ein PUT-Request einer Drittsoftware wird von einer Instanz von *OrchidSoftwareDaemon* verarbeitet. Die zu speichernde Instruktion wird aus der Anfrage ausgelesen, in der Datenbank gespeichert und eine Erfolgsmeldung an die Drittsoftware übermittelt.
Datenstrukturen	*Orchid.Core.Requests.OrchidHttpRequest, Orchid.Core.HttpRequestType, Orchid.Core.Data.Instruction*
Beteiligte	*Orchid.Core.Daemons.OrchidSoftwareDaemon, Orchid.Core.Data.MySqlAdapter*
Kategorie	Musskriterium

Tabelle 19 - FS150: Client- Instruktion speichern

/FS160/	Client-Informationen abrufen
Vorbedingung	/FS100/, Request-Typ „SEARCH“, angenommen durch Software-Server
Erfolgsfall	Informationen über den angegebenen ORCHID Client wurden aus der Datenbank ausgelesen und an Drittsoftware übermittelt
Fehlerfall	Informationen konnten nicht ermittelt werden, Fehlermeldung an Drittsoftware
Beschreibung	Ein SEARCH-Request einer Drittsoftware wird von einer Instanz von *OrchidSoftwareDaemon* verarbeitet. Die Informationen über den Client werden aus der Datenbank gelesen und an die Drittsoftware übermittelt.
Datenstrukturen	*Orchid.Core.Requests.OrchidHttpRequest, Orchid.Core.HttpRequestType*
Beteiligte	*Orchid.Core.Daemons.OrchidSoftwareDaemon, Orchid.Core.Data.MySqlAdapter*
Kategorie	Musskriterium

Tabelle 20 - FS160: Client-Informationen abrufen

/FS170/	Ungültige oder fehlerhafte Anfrage behandeln
Vorbedingung	/FS100/
Erfolgsfall	Ungültige oder fehlerhafte Anfrage wird nicht verarbeitet, Fehlermeldung wird an Verbindungspartner übermittelt
Fehlerfall	Anfrage kann nicht bearbeitet werden, Fehlermeldung an Verbindungspartner
Beschreibung	Ungültige, unbekannte oder fehlerhafte HTTP-Anfragen werden nicht verarbeitet, sondern mit einer Fehlermeldung an den Verbindungspartner beantwortet.
Datenstrukturen	*Orchid.Core.Requests.OrchidHttpRequest, Orchid.Core.HttpRequestType*
Beteiligte	*Orchid.Core.Daemons.OrchidHardwareDaemon oder Orchid.Core.Daemons.OrchidSoftwareDaemon*
Kategorie	Musskriterium

Tabelle 21 - FS170: Ungültige oder fehlerhafte Anfrage behandeln

/FS180/	Nachricht speichern (gesendet durch Drittsoftware)
Vorbedingung	/FS100/, Request-Typ „POST“, angenommen durch Software-Server
Erfolgsfall	Nachricht wurde gespeichert und kann von einem Client abgerufen werden.
Fehlerfall	Nachricht wurde nicht gespeichert, Fehlermeldung an Verbindungspartner
Beschreibung	Ein POST-Request einer Drittsoftware wird durch eine Instanz von *OrchidSoftwareDaemon* verarbeitet. Die Nachricht wird der Anfrage entnommen und in der Datenbank gespeichert.
Datenstrukturen	*Orchid.Core.Requests.OrchidHttpRequest, Orchid.Core.HttpRequestType, Orchid.Core.Data.Message*
Beteiligte	*Orchid.Core.Daemons.OrchidSoftwareDaemon, Orchid.Core.Data.MySqlAdapter*
Kategorie	Wunschkriterium

Tabelle 22 - FS180: Nachricht speichern (Drittsoftware)

/FS190/	Nachricht speichern (gesendet durch ORCHID Client)
Vorbedingung	/FS100/, Request-Typ „POST“, angenommen durch Hardware-Server
Erfolgsfall	Nachricht wurde gespeichert und kann abgerufen werden.
Fehlerfall	Nachricht wurde nicht gespeichert, Fehlermeldung an Verbindungspartner
Beschreibung	Ein POST-Request eines Clients wird durch eine Instanz von *OrchidHardwareDaemon* verarbeitet. Die notwendigen Informationen über die zu speichernde Nachricht werden der Anfrage entnommen und die neue Nachricht in der Datenbank gespeichert.
Datenstrukturen	*Orchid.Core.Requests.OrchidHttpRequest, Orchid.Core.HttpRequestType, Orchid.Core.Data.Message*
Beteiligte	*Orchid.Core.Daemons.OrchidHardwareDaemon, Orchid.Core.Data.MySqlAdapter*
Kategorie	Wunschkriterium

Tabelle 23 - FS190: Nachricht speichern (Client)

/FS200/	Nachricht an ORCHID Client übermitteln
Vorbedingung	/FS100/, Request-Typ „SEARCH“, angenommen durch Hardware-Server
Erfolgsfall	Nachricht wird an den anfragenden Client übermittelt.
Fehlerfall	Nachricht wurde nicht übermittelt, Fehlermeldung an Client
Beschreibung	Ein SEARCH-Request eines Clients wird durch eine Instanz von `OrchidHardwareDaemon` verarbeitet. Die notwendigen Informationen über eine zu versendende Nachricht werden der Anfrage entnommen, die entsprechende Nachricht aus der Datenbank ausgelesen und an den anfragenden ORCHID Client übermittelt.
Datenstrukturen	`Orchid.Core.Requests.OrchidHttpRequest, Orchid.Core.HttpRequestType, Orchid.Core.Data.Message`
Beteiligte	`Orchid.Core.Daemons.OrchidHardwareDaemon, Orchid.Core.Data.MySqlAdapter`
Kategorie	Wunschkriterium

Tabelle 24 - FS200: Nachricht an ORCHID Client übermitteln

/FS210/	Berechtigungen für ORCHID Client abrufen
Vorbedingung	-
Erfolgsfall	Berechtigungen für einen ORCHID Client wurden aus der Datenbank abgerufen und können verwendet werden.
Fehlerfall	Berechtigungen können nicht verwendet werden, jeder Client hat alle Berechtigungen.
Beschreibung	Für jeden ORCHID Client, der einem ORCHID Server bekannt ist, existieren Berechtigungsinformationen, die angeben, was dieser Client tun darf und was nicht. Diese Berechtigungen (`Credentials`) sagen aus, ob ein Client • Daten auf den Server laden darf • Instruktionen vom Server abrufen darf • Nachrichten empfangen darf • Nachrichten senden darf • Instruktionen für andere Clients auf den Server laden darf • Instruktionen für sich selbst vom Server abrufen darf
Datenstrukturen	`Orchid.Core.Requests.OrchidHttpRequest, Orchid.Core.HttpRequestType, Orchid.Core.Data.Credentials`
Beteiligte	`Orchid.Core.Daemons.OrchidHardwareDaemon Orchid.Core.Data.MySqlAdapter`
Kategorie	Wunschkriterium

Tabelle 25 - FS210: Berechtigungen für ORCHID Client abrufen

/FS220/	Client-Instruktion speichern (gesendet durch ORCHID Client)
Vorbedingung	/FS100/, Request-Typ „CONNECT“, angenommen durch Hardware-Server
Erfolgsfall	Instruktion wurde gespeichert und kann von einem Client abgerufen werden.
Fehlerfall	Anfrage kann nicht bearbeitet werden, Fehlermeldung an Verbindungspartner, Instruktion wurde nicht gespeichert.
Beschreibung	Ein CONNECT-Request eines Clients wird durch eine Instanz der Klasse `OrchidHardwareDaemon` verarbeitet. Aus der Anfrage werden die notwendigen Daten ausgelesen und eine neue Instruktion in der Datenbank gespeichert.
Datenstrukturen	`Orchid.Core.Requests.OrchidHttpRequest, Orchid.Core.HttpRequestType, Orchid.Core.Data.Instruction`
Beteiligte	`Orchid.Core.Daemons.OrchidHardwareDaemon, Orchid.Core.Data.MySqlAdapter`
Kategorie	Wunschkriterium

Tabelle 26 - FS220: Client-Instruktion speichern (Client)

/FS230/	Developer Key prüfen
Vorbedingung	/FS90/
Erfolgsfall	Übertragener Developer Key kann mit dem auf dem Server gespeicherten Key verglichen werden.
Fehlerfall	Developer Key kann nicht geprüft werden, Abbruch der Anfragebehandlung.
Beschreibung	Ein Client oder eine Drittsoftware übermittelt in einer Anfrage an den Server eine Zeichenkette, die zur Identifizierung als erlaubter Kommunikationspartner gilt.
Datenstrukturen	`Orchid.Core.Requests.OrchidHttpRequest`
Beteiligte	`Orchid.Core.Daemons.OrchidHardwareDaemon` oder `Orchid.Core.Daemons.OrchidSoftwareDaemon`
Kategorie	Wunschkriterium

Tabelle 27 - /FS230/ - Developer Key prüfen

4.4.2 Funktionen der Client-Bibliothek

Die folgende Beschreibung der Funktionen der Client-Bibliothek verzichtet auf die Angabe von Beteiligten, da dies stets der ausführende Sketch, die Arduino-Ethernet-Library (vgl. [21]) und die Client-Bibliothek sind (vgl. 5.1.2).

/FC10/	ORCHID Client-Instanz erzeugen
Vorbedingung	Arduino bereit
Erfolgsfall	Eine Instanz der Client-Bibliothek ist bereit und initialisiert
Fehlerfall	Client kann nicht kommunizieren
Beschreibung	Eine Instanz des ORCHID Client wird erzeugt und initialisiert. Ihr werden verschiedene Einstellungen mitgeteilt (z.B. MAC-Adresse, IP-Adresse, Server-Adresse, ID des Clients, etc.).
Kategorie	Musskriterium

Tabelle 28 - FC10: ORCHID Client-Instanz erzeugen

/FC20/	Sensorwert an ORCHID Server übermitteln
Vorbedingung	/FC10/, /FC80/
Erfolgsfall	Sensorwert wurde an Server übermittelt und in der Datenbank gespeichert
Fehlerfall	Sensorwert kann nicht hochgeladen und/oder nicht in der Datenbank gespeichert werden.
Beschreibung	Die Client-Bibliothek verfügt über zahlreiche Methoden, Daten an einen ORCHID Server zu übermitteln (siehe Unterfunktionen dieser Funktion). Eine Verbindung zum Server wird aufgebaut, eine entsprechende HTTP-Anfrage (PUT-Request) gesendet und die Antwort ausgelesen.
Kategorie	Musskriterium

Tabelle 29 - FC20: Sensorwert an ORCHID Server übermitteln

/FC20a/	Sensorwert an ORCHID Server übermitteln (Integer)
Vorbedingung	/FC10/

Tabelle 30 - FC20a: Integer übermitteln

/FC20b/	Sensorwert an ORCHID Server übermitteln (Float)
Vorbedingung	/FC10/

Tabelle 31 – FC20b: Float übermitteln

/FC20c/	Sensorwert an ORCHID Server übermitteln (Double)
Vorbedingung	/FC10/

Tabelle 32 - FC20c: Double übermitteln

/FC20d/	Sensorwert an ORCHID Server übermitteln (Unsigned Integer)
Vorbedingung	/FC10/

Tabelle 33 - FC20d: Unsigned Integer übermitteln

/FC20e/	Sensorwert an ORCHID Server übermitteln (Character)
Vorbedingung	/FC10/

Tabelle 34 - FC20e: Character übermitteln

/FC20f/	Sensorwert an ORCHID Server übermitteln (Boolean)
Vorbedingung	/FC10/

Tabelle 35 - FC20f: Boolean übermitteln

/FC20g/	Sensorwert an ORCHID Server übermitteln (Zeichenkette)
Vorbedingung	/FC10/
Anmerkung	Es handelt sich intern um einen `char*`, also einen Zeiger auf eine Kette von Zeichen, weshalb die Bezeichnung *„String“* nicht verwendet wurde.

Tabelle 36 - FC20g: Zeichenkette übermitteln

/FC20h/	Sensorwert an ORCHID Server übermitteln (Byte)
Vorbedingung	/FC10/
Anmerkung	Intern wird der Datentyp `uint8_t` verwendet, der in der Header-Datei `inttypes.h` definiert ist.

Tabelle 37 - FC20h: Byte übermitteln

/FC20i/	Sensorwert an ORCHID Server übermitteln (Long)
Vorbedingung	/FC10/

Tabelle 38 - FC20i: Long übermitteln

/FC20j/	Sensorwert an ORCHID Server übermitteln (Unsigned Long)
Vorbedingung	/FC10/

Tabelle 39 - FC20j: Unsigned Long übermitteln

/FC30/	Heartbeat an ORCHID Server senden
Vorbedingung	/FC10/
Erfolgsfall	Heartbeat-Anfrage wurde an ORCHID Server übermittelt, der Datensatz des sendenden Clients in der Datenbank aktualisiert.
Fehlerfall	Heartbeat-Anfrage kann nicht gesendet, Datenbankeintrag nicht aktualisiert werden.
Beschreibung	Ein ORCHID Client kann einen Heartbeat-Request (TRACE-Request) an einen ORCHID Server übermitteln, um diesem mitzuteilen, dass er bereit ist.
Kategorie	Musskriterium

Tabelle 40 - FC30: Heartbeat an ORCHID Server senden

/FC40/	Instruktion für Aktor abrufen
Vorbedingung	/FC10/, /FC80/
Erfolgsfall	Die nächste unbearbeitete Instruktion für einen an den sendenden Client angeschlossenen Aktor wurde vom Server abgerufen und liegt in einer Datenstruktur vor.
Fehlerfall	Die nächste unbearbeitete Instruktion kann nicht abgerufen werden.
Beschreibung	Ein ORCHID Client kann unter Angabe seiner eigenen ID und der ID des Aktors, für den eine Instruktion abgerufen werden soll, einen GET-Request an einen ORCHID Server senden. Dieser Request wird im Erfolgsfall unter Angabe der nächsten Instruktion beantwortet (im Fehlerfall oder falls keine Instruktion vorliegt, variiert die Antwort).
Datenstrukturen	*Instruction*
Kategorie	Musskriterium

Tabelle 41 - FC40: Instruktion für Aktor abrufen

/FC50/	Instruktion für Aktor ausführen
Vorbedingung	/FC10/, /FC80/
Erfolgsfall	Die nächste unbearbeitete Instruktion für einen an den sendenden Client angeschlossenen Aktor wird direkt ausgeführt.
Fehlerfall	Die nächste unbearbeitete Instruktion kann nicht ausgeführt werden.
Beschreibung	Wie in /FC40/ beschrieben, wird die nächste unbearbeitete Instruktion vom Server abgerufen, in dieser Funktion jedoch direkt ausgeführt.
Datenstrukturen	*Instruction*
Kategorie	Musskriterium

Tabelle 42 - FC50: Instruktion für Aktor ausführen

/FC60/	Anfrage an ORCHID Server senden
Vorbedingung	/FC10/
Erfolgsfall	Eine HTTP-Anfrage wurde an einen ORCHID Server gesendet
Fehlerfall	Die gewünschte HTTP-Anfrage kann nicht gesendet werden
Beschreibung	Die Client-Bibliothek nutzt die Arduino-Ethernet-Library, um HTTP-Anfragen an einen ORCHID Server zu senden (die Ethernet-Library wird über die an die Client-Bibliothek übergebenen Parameter wie IP-Adresse, Server-Adresse, etc. konfiguriert).
Datenstrukturen	*Client*
Kategorie	Musskriterium

Tabelle 43 - FC60: Anfrage an ORCHID Server senden

/FC70/	Auf eingehende Server-Antwort warten
Vorbedingung	/FC60/
Erfolgsfall	Die Ausführung des Sketches wird unterbrochen, bis eine Antwort auf eine gesendete Anfrage eingeht.
Fehlerfall	Der Prozess, der mit der ausgehenden Anfrage eingeleitet wurde, kann nicht durchgeführt werden.
Beschreibung	Nachdem eine Anfrage an einen ORCHID Server gesendet wurde, wird auf eine Antwort des Servers gewartet und das Programm/der Sketch unterbrochen, bis diese Antwort eintrifft oder ein vorher definierter Zeitraum überschritten wurde (Timeout).
Datenstrukturen	`Client`
Kategorie	Musskriterium

Tabelle 44 - FC70: Auf eingehende Server-Antwort warten

/FC80/	Eingehende Server-Antwort auslesen
Vorbedingung	/FC60/
Erfolgsfall	Eine Antwort, die ein ORCHID Server auf eine ausgehende HTTP-Anfrage gegeben hat, wird ausgelesen und kann verwendet werden.
Fehlerfall	Die eingehende Antwort kann nicht gelesen und daher auch nicht analysiert bzw. weiter verwendet werden.
Beschreibung	Eine eingehende Antwort eines Servers liegt als Strom von Bytes vor, die nach und nach ausgelesen und zu einer Nachricht zusammengesetzt werden. Aus den so ermittelten Daten können Informationen über den Erfolg oder Misserfolg der Anfrage, die die Antwort des Servers ausgelöst hat, entnommen werden.
Datenstrukturen	`Client`
Kategorie	Musskriterium

Tabelle 45 - FC80: Eingehende Server-Antwort auslesen

/FC90/	*Nachricht auf ORCHID Server hochladen*
Vorbedingung	/FC60/
Erfolgsfall	Die Nachricht (für einen anderen ORCHID Client) wurde auf den Server geladen und kann abgerufen werden.
Fehlerfall	Die Nachricht kann nicht auf den Server geladen werden.
Beschreibung	Ein ORCHID Client sendet eine Nachricht über einen POST-Request an einen ORCHID Server.
Datenstrukturen	`Client`
Kategorie	Wunschkriterium

Tabelle 46 - FC90: Nachricht auf ORCHID Server hochladen

/FC100/	*Nachricht von ORCHID Server abrufen*
Vorbedingung	/FC60/
Erfolgsfall	Die Nachricht wurde abgerufen und kann verwendet werden.
Fehlerfall	Die Nachricht kann nicht abgerufen werden.
Beschreibung	Ein ORCHID Client ruft eine Nachricht über einen SEARCH-Request an einen ORCHID Server ab.
Datenstrukturen	`Client`
Kategorie	Wunschkriterium

Tabelle 47 - FC100: Nachricht von ORCHID Server abrufen

/FC110/	*Instruktion an ORCHID Server senden*
Vorbedingung	/FC60/
Erfolgsfall	Die Instruktion wurde auf den ORCHID Server geladen und kann abgerufen werden.
Fehlerfall	Die Instruktion wurde nicht auf den ORCHID Server geladen.
Beschreibung	Ein ORCHID Client kann eine Instruktion für einen anderen Client über eine CONNECT-Anfrage an einen ORCHID Server senden.
Datenstrukturen	`Client`
Kategorie	Wunschkriterium

Tabelle 48 - FC110: Instruktion an ORCHID Server senden

4.5 Produktdaten

Eine Hauptaufgabe des ORCHID Toolkits ist es, von ORCHID Clients gemessene Daten zu speichern und abzurufen. Die folgenden Tabellen zeigen, mit welchen Daten die Server-Komponente und die Client-Bibliothek arbeiten und geben, falls notwendig, wichtige zusätzliche Hinweise zu diesen Daten.

4.5.1 Server-Komponente

Für die Server-Komponente gilt, dass die Menge an gespeicherten Daten nur durch den physikalischen Speicher des ausführenden Computersystems begrenzt ist. Theoretisch können beliebig viele Datensätze gespeichert werden. Daten, die durch ein Wunschkriterium (vgl. 4.1.2) verwaltet werden müssen, sind in ihrem Titel kursiv hervorgehoben.

/DS10/	**Aktoren**
Inhalt	• Name • Eindeutige ID (GUID) • Beschreibung
Anmerkungen	Gespeichert in der Datenbanktabelle „`actors`“.

Tabelle 49 - DS10: Aktoren

/DS20/	**Sensoren**
Inhalt	• Name • Eindeutige ID (GUID) • Beschreibung
Anmerkungen	Gespeichert in der Datenbanktabelle „`sensors`“.

Tabelle 50 - DS20: Sensoren

/DS30/	Arduinos
Inhalt	• Name • Eindeutige ID (GUID) • Numerische Datenbank-ID • Beschreibung • *Aktoren* • *Sensoren* • *Geolocation* • Zeitstempel für ersten Kontakt • Zeitstempel für letzten Kontakt
Anmerkungen	Gespeichert in der Datenbanktabelle „`arduinos`“.

Tabelle 51 – DS30: Arduinos

/DS40/	*Berechtigungen*
Inhalt	• Eindeutige ID (GUID) des betroffenen *Arduinos* • Sendeberechtigung für Nachrichten • Empfangsberechtigung für Nachrichten • Berechtigung zum Hochladen von Sensordaten • Empfangsberechtigung für Instruktionen • Berechtigung zum Hochladen von Instruktionen
Anmerkungen	Gespeichert in der Datenbanktabelle „`credentials`“.

Tabelle 52 – DS40: Berechtigungen

/DS50/	Geolocations (Position auf der Erde)
Inhalt	• Längengrad • Breitengrad
Anmerkungen	Gespeichert als Teil des Datensatzes eines Arduinos, vgl. /DS30/.

Tabelle 53 – DS50: Geolocations

/DS60/	Sensordaten
Inhalt	• Eindeutige ID (GUID) des *Sensors*, der die Daten gemessen hat • Textuelle Repräsentation des Datentyps der Sensordaten • Textuelle Repräsentation der gemessenen Daten • Zeitpunkt der Messung
Anmerkungen	Gespeichert in der Datenbanktabelle „`sensor_data`“.

Tabelle 54 - DS60: Sensordaten

/DS70/	Instruktionen
Inhalt	• Numerische Datenbank-ID • Eindeutige ID (GUID) des Arduinos, der die Instruktion ausführen soll • Eindeutige ID (GUID) des Aktors, für den die Instruktion bestimmt ist • Numerischer Wert der durchzuführenden Aktion (vgl. 8.4.4) • Wert für die Aktion • Zeitpunkt, zu dem die Instruktion erzeugt wurde • Bearbeitungsstand (offen/erledigt) • Numerischer Wert, der den Typ des Werts der Aktion anzeigt • Index des Arduino-I/O-Pins, an dem die Aktion auszuführen ist • Prioritätswert
Anmerkungen	Gespeichert in der Datenbanktabelle „`instructions`“.

Tabelle 55 – DS70: Instruktionen

/DS80/	*Nachrichten*
Inhalt	• Eindeutige ID (GUID) des Absenders • Eindeutige ID (GUID) des Empfängers • Nachrichtentext • Zeitpunkt, zu dem die Nachricht erzeugt wurde • Bearbeitungsstand (neu/empfangen) • Prioritätswert
Anmerkungen	Gespeichert in der Datenbanktabelle „`messages`“.

Tabelle 56 - DS80: Nachrichten

4.5.2 Client-Bibliothek

Die Client-Bibliothek ist nicht in der Lage, Daten dauerhaft zu speichern (dies ist auch nicht notwendig). Alle hier genannten Daten sind also flüchtig und existieren nur zur Laufzeit.

/DC10/	Instruktionen
Inhalt	• Eindeutige ID (GUID) des Aktors • Numerischer Wert für die durchzuführende Aktion • Numerischer Wert für den Datentyp des Werts • Wert der Aktion • I/O-Pin, an dem die Aktion durchgeführt werden soll
Anmerkungen	Numerische Werte sind als Konstanten definiert und können entsprechend leicht verwendet werden.

Tabelle 57 - DC10: Instruktionen

4.6 Produktleistung

Die in diesem Abschnitt beschriebenen Leistungsmerkmale des Toolkits sind in zwei Kategorien unterteilt: Server-Leistung und Client-Leistung. Leistungen, die die Server-Komponente erfüllen muss, sind mit $/LS_n/$ gekennzeichnet, Client-Leistungen mit $/LC_n/$.

4.6.1 Server-Komponente

Die folgenden Tabellen zeigen die Leistungsmerkmale, die von der Server-Komponente erfüllt werden müssen.

/LS10/	Antwortzeiten
Beschreibung	Die Server-Komponente muss – bei Vorhandensein entsprechend leistungsfähiger Hardware – eingehende HTTP-Anfragen zeitnah behandeln und möglichst schnell eine Antwort senden.
Metrik	Die durchschnittliche Zeit zwischen eingehender Anfrage und ausgehender Antwort darf eine Sekunde nicht überschreiten.
Bemerkung	Da die Übertragungsgeschwindigkeit von Anfragen und Antworten direkt mit der Geschwindigkeit des Netzwerks abhängt, gilt die für /LS10/ angegebene Metrik für das Szenario, dass sich der Server und die Clients im gleichen, lokalen Netzwerk befinden und die verwendete Hardware mindestens der des Entwicklungssystems entspricht (vgl. 6.2).

Tabelle 58 - LS10: Antwortzeiten

/LS20/	Betriebsdauer
Beschreibung	Die Server-Komponente ist für den Dauerbetrieb ausgelegt, längere Betriebszeiten haben keinen negativen Einfluss auf die Leistung der Komponente.
Metrik	Die Betriebsdauer wird in Tagen, Stunden, Minuten und Sekunden gemessen.
Bemerkung	-

Tabelle 59 - LS20: Betriebsdauer

/LS30/	Belastbarkeit
Beschreibung	Obwohl das ORCHID Toolkit nicht für Szenarien mit hoher Belastungsdichte ausgelegt ist, ist die Server-Komponente dennoch in der Lage, auch bei erhöhtem Datenaufkommen ihre Leistungsfähigkeit aufrecht zu erhalten.
Metrik	Die Server-Komponente muss im Durchschnitt mindestens 5 Anfragen pro Sekunde verarbeiten und beantworten können.
Bemerkung	Da die Übertragungsgeschwindigkeit von Anfragen und Antworten direkt mit der Geschwindigkeit des Netzwerks abhängt, gilt die für /LS30/ angegebene Metrik für das Szenario, dass sich der Server und die Clients im gleichen, lokalen Netzwerk befinden und die verwendete Hardware mindestens der des Entwicklungssystems entspricht (vgl. 6.2). Ausgenommen von der Mindestanzahl sind Anfragen, die zum Abruf von Sensordaten führen, da diese in verschieden langen Antwortzeiten resultieren können, je nachdem, wie viele Datensätze abgerufen werden sollen.

Tabelle 60 - LS30: Belastbarkeit

/LS40/	Datengenauigkeit
Beschreibung	Die Server-Komponente darf die Genauigkeit ein- oder ausgehender Daten nicht verändern.
Metrik	Ganzzahlen werden auf 1 genau berechnet, Fließkommazahlen auf mindestens 2 Stellen nach dem Komma genau. Datums- und Zeitangaben werden auf die Sekunde genau berechnet.
Bemerkung	Werden Daten angeliefert, die weniger genau als beschrieben sind, werden diese nicht verändert; ist eine Veränderung zwingend notwendig, wird als verändernder Wert 0 genutzt (eine Zeitangabe, für die nur ein Datum bereitgestellt wird, wird also um die Zeit 00:00:00 erweitert).

Tabelle 61 - LS40: Datengenauigkeit

/LS50/	Robustheit
Beschreibung	Sie Server-Komponente verursacht keine Programmabstürze. Fehler werden aufgefangen und gemeldet, der reguläre Betrieb danach (falls möglich) wiederhergestellt. Ist eine Wiederherstellung nicht möglich, wird die Ausführung der Server-Komponente mit einer Fehlermeldung beendet, es kommt aber nicht zu einem Programmabsturz.
Metrik	-
Bemerkung	Angriffe auf die Server-Komponente (z.B. *Denial of Service*, siehe dazu [22], S. 274 ff.) werden nicht als regulärer Betrieb eingestuft.

Tabelle 62 - LS50: Robustheit

4.6.2 Client-Bibliothek

Die Client-Bibliothek muss die in den folgenden Tabellen dargestellten Leistungsmerkmale erfüllen können.

/LC10/	Betriebsdauer
Beschreibung	Die Client-Bibliothek ist für den Dauerbetrieb ausgelegt, längere Betriebszeiten beeinflussen die Ausführung der Bibliothek nicht negativ.
Metrik	Die Betriebsdauer wird in Tagen, Stunden, Minuten und Sekunden gemessen.
Bemerkung	Die Betriebsdauer hängt direkt vom verwendeten Arduino Microcontroller ab und dessen Stromversorgung ab.

Tabelle 63 - LC10: Betriebsdauer

/LC20/	Datengenauigkeit
Beschreibung	Die Client-Bibliothek verändert die Genauigkeit ein- und ausgehender Daten nicht.
Metrik	Ganzzahlen werden auf 1 genau berechnet, Fließkommazahlen auf mindestens 2 Stellen nach dem Komma genau.
Bemerkung	Werden Daten angeliefert, die weniger genau als beschrieben sind, werden diese nicht verändert.

Tabelle 64 - LC20: Datengenauigkeit

/LC30/	Robustheit
Beschreibung	Die Client-Bibliothek stürzt im regulären Betrieb nicht ab und verursacht auch keine Abstürze des sie verwendenden Sketches (siehe dazu auch 6.6)
Metrik	-
Bemerkung	Die Unterbrechung der Stromversorgung, der Netzwerkverbindung oder eines Schaltkreises werden nicht als regulärer Betrieb betrachtet.

Tabelle 65 - LC30: Robustheit

4.7 Qualitätsanforderungen

In diesem Abschnitt der Arbeit werden technische, wirtschaftliche und ergonomische Anforderungen an die Softwarequalität des Toolkits beschrieben. Falls nicht anders angegeben, beziehen sich die hier geschilderten Anforderungen sowohl auf die Server-Komponente, als auch auf die Client-Bibliothek. Ist in diesem Abschnitt von „Software“ die Rede, bezieht sich diese Aussage auf die im Rahmen dieser Arbeit entwickelte Software, nicht auf Drittsoftware, die das ORCHID Toolkit verwendet.

4.7.1 Technische Qualitätsanforderungen

Durch den konsequenten Einsatz von objektorientierten Programmiertechniken und Entwurfsmustern (hauptsächlich in der Server-Komponente), entsteht eine hoch funktionale, leicht wart- und anpassbare Software, die darüber hinaus auch von Dritten leicht erfasst und eingesetzt werden kann. Neue Funktionen können problemlos hinzugefügt werden, ohne den Kern der Software ändern zu müssen.

Fehler, die durch einen Benutzer (Entwickler, der das Toolkit einsetzt) gemacht werden, werden nach Möglichkeit abgefangen und korrigiert. Fehlerhafte Benutzereingaben führen nicht zu einem kritischen Programmzustand oder zu einem Absturz der Software (falls nicht anders möglich, wird die Software neu gestartet, stürzt aber nicht ab). Dieser Grundsatz wird auch auf alle anderen Fehler angewendet, die im regulären Betrieb auftreten können.

Kommt es zur Laufzeit der Software dennoch zu einem unvorhergesehenen Fehler und führt dieser Fehler zur Beendigung der Software, gehen dabei keine Daten verloren (Ausnahme sind Daten, die in dem Moment verarbeitet werden, in dem die Software zwangsweise beendet wird). Ein Neustart der Anwendung versetzt sie in exakt den Zustand, in dem sie sich vor der Beendigung befand.

Eine Installation der Software ist nicht nötig. Wechselwirkungen mit anderen auf einem ausführenden System laufenden Programmen und Systemen sind nicht zu erwarten und können, sollten sie dennoch auftreten, über die Konfiguration der Software behoben werden (dies gilt hauptsächlich für die Server-Komponente).

4.7.2 Wirtschaftliche Qualitätsanforderungen

Die entwickelte Software ist sehr flexibel und kann mit geringem Aufwand an gänzlich neue Aufgaben angepasst werden. Durch die Verwendung von Entwurfsmustern, objektorientierten Programmiertechniken und einem dynamischen Typeloading-System ist die Anpassung an neue Aufgaben auch ohne Veränderung der Kernsoftware möglich (gilt für die Server-Komponente, vgl. 6.3.2).

Der Einsatz von HTTP zur Kommunikation zwischen den einzelnen Bestandteilen des ORCHID Toolkits erlaubt es grundsätzlich jedem Entwickler, es für die Integration von Hardware in ein eigenes Softwareprojekt zu nutzen, da jede moderne Programmiersprache über die Möglichkeit verfügt, HTTP-Anfragen zu senden und Server-Antworten zu empfangen.

Das Toolkit ist nicht an ein Betriebssystem gebunden, sondern kann auf allen Systemen ausgeführt werden, die es erlauben, das Mono Framework zu installieren und auszuführen (vgl. 4.3.1). Zusätzlich wird das Toolkit unter einer sehr flexiblen Open Source Lizenz veröffentlicht (vgl. 4.2.3), um für die Verwendung in möglichst vielen Projekten geeignet zu sein. Die Lizenz erhöht die Wiederverwendbarkeit und Flexibilität der Software, da auch Weiterentwicklungen und Veränderungen unter die gleiche oder eine ähnliche Lizenz gestellt werden müssen und so einer breiten Zielgruppe zur Verfügung stehen.

4.7.3 Software-Ergonomie

Die saubere semantische Trennung von Klassen innerhalb des Toolkits führt zu einem erleichterten Verständnis sowohl der Programmstrukturen, als auch der Programmabläufe. Meldungen an Benutzer und Kommunikationspartner sind aussagekräftig (vgl. 5.8 und 5.5.1.3) und leicht verständlich.

Als natürliche Sprache, die innerhalb des ORCHID Toolkits zum Einsatz kommt, wurde Englisch gewählt. Dies gilt sowohl für Meldungen der Toolkit-Komponenten, als auch für Kommentare und Namen innerhalb des Quellcodes. Diese Entscheidung führt dazu, dass nahezu jeder Softwareentwickler in die Lage versetzt wird, die Software einsetzen und verstehen zu können.

Alle gespeicherten Daten (inkl. Einstellungen) liegen in menschenlesbarer Form vor (gilt für die Server-Komponente).

Die hohe Konfigurierbarkeit der Server-Komponente führt darüber hinaus dazu, dass zahlreiche Aspekte des Toolkits an die Vorstellungen eines Entwicklers, der es einsetzt, angepasst werden können, ohne den Code der Software verändern zu müssen.

5 Systementwurf

Im Folgenden wird mit der grundlegenden Struktur des Toolkits die Basis für die Implementierung der Software, die im Rahmen dieser Arbeit entwickelt wurde, beschrieben. Der Fokus der Beschreibung liegt dabei insbesondere auf strukturellen Besonderheiten, die für die Verwendung oder die Weiterentwicklung des Toolkits wichtig sind.

5.1 Übersicht

Wie bereits erwähnt, besteht das ORCHID Toolkit aus zwei Hauptbestandteilen: der Server-Komponente und der Client-Bibliothek. Für beide Bestandteile ist zusätzlich eine exemplarische Referenzimplementierung enthalten. Darüber hinaus wurde im Rahmen dieser Arbeit eine Drittsoftware entwickelt, die aber nicht Teil des eigentlichen Toolkits ist, sondern nur als Proof of Concept gilt und deshalb an dieser Stelle nicht betrachtet wird.

Im Folgenden werden lediglich die eigentlichen Komponenten des Toolkits beschrieben, Details zu den Referenzimplementierungen und der Drittsoftware können den Abschnitten 7, 8 und 10 entnommen werden.

5.1.1 Projektstruktur Server-Komponente

Die gesamte Funktionalität, die einem ORCHID Server durch `Orchid.Core` zur Verfügung gestellt wird, befindet sich in einer einzigen Assembly. Obwohl dieser Ansatz dazu führt, dass diese eine Assembly sehr viele Klassen enthält, erscheint er unter dem Gesichtspunkt der einfachen Verwendung des Toolkits jedoch geeigneter als die Verteilung der Klassen auf mehrere Assemblies. Ein Entwickler, der das Toolkit einsetzt, muss lediglich eine einzige Datei in sein Server-Projekt integrieren, damit die gesamte Funktionalität des ORCHID Toolkits genutzt werden kann. Um die Übersicht innerhalb der Assembly `Orchid.Core` dennoch nicht zu gefährden, ist sie in zahlreiche semantisch sauber voneinander getrennte Namespaces aufgeteilt, die im weiteren Verlauf dieses Abschnitts im Detail erläutert werden.

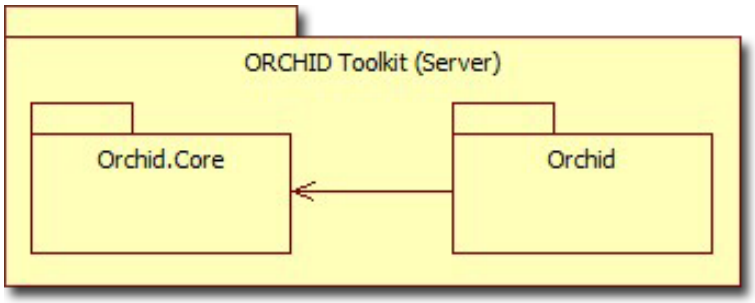

Abbildung 10 - Projektstruktur der Server-Komponente

Die obige Abbildung 10 zeigt die Struktur der Server-Komponente zusammen mit der Referenzimplementierung (`Orchid`) anhand einer an die UML-Notation angelehnten Grafik. Die Abhängigkeit der Assembly `Orchid` von der Assembly `Orchid.Core` ist einseitig, d.h. `Orchid.Core` kann auch ohne `Orchid` eingesetzt werden, das Gegenteil ist nicht möglich.

Um durch die erhöhte Anzahl von Klassen in der Assembly `Orchid.Core` die Übersicht dennoch nicht zu gefährden, wurden die Klassen in semantisch voneinander getrennte Namespaces aufgeteilt. Die folgende Abbildung zeigt die wichtigsten dieser Namespaces in einem (aus Gründen der Übersicht nicht vollständigen) UML-Paketdiagramm.

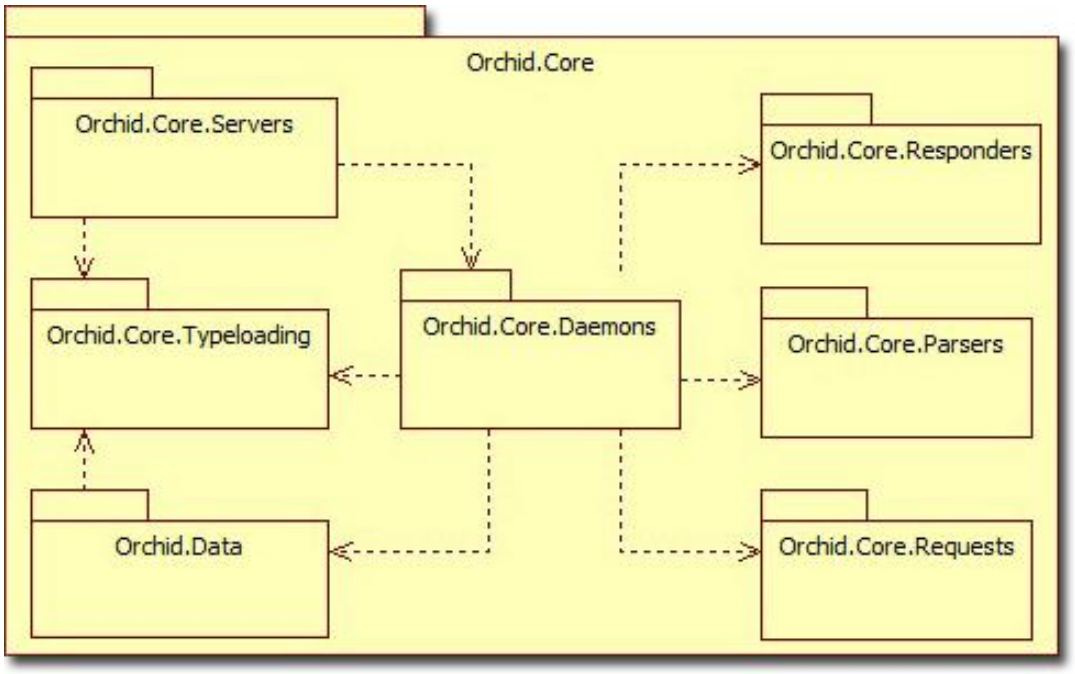

Abbildung 11 - Namespaces in "Orchid.Core"

Aus der obigen Abbildung 11 wird bereits ohne weitere Erläuterung die Hierarchie innerhalb der Assembly deutlich: Die Klassen in `Orchid.Core.Daemons` sind die zentralen Klassen der Server-Komponente. Sie verwenden Klassen aus beinahe allen anderen Namespaces, um die ihnen zugeteilten Aufgaben zu erledigen, und werden ihrerseits von den Klassen in `Orchid.Core.Servers` verwendet.

Die einzelnen Namespaces aus `Orchid.Core`, auch diejenigen, die nicht in Abbildung 11 sichtbar sind, werden im Folgenden (alphabetisch sortiert) beschrieben. Eine detaillierte Beschreibung einzelner Klassen befindet sich im Abschnitt 5.6.

5.1.1.1 Orchid.Core

Dieser Namespace, der den gleichen Namen wie die ihn enthaltende Assembly trägt, enthält die Hauptklasse `Bloom` der Server-Komponente, die in Implementierungen eines ORCHID-Servers genutzt werden kann, sowie zusätzliche Hilfsklassen.

5.1.1.2 Orchid.Core.Configuration

`Orchid.Core.Configuration` enthält alle zur Verwendung des Konfigurationssystems notwendigen Klassen. Es handelt sich um einen zentralen Namespace, auf den aus den meisten anderen Namespaces zugegriffen wird. Aus diesem Grund ist er in Abbildung 11 nicht enthalten.

5.1.1.3 Orchid.Core.Daemons

Alle Daemon-Klassen (und deren abstrakte Basisklassen) sind in `Orchid.Core.Daemons` enthalten. Zusätzlich enthält der Namespace abstrakte und konkrete Fabrikklassen für Daemon-Instanzen.

5.1.1.4 Orchid.Core.Data

Dieser Namespace enthält Datenhaltungsklassen für alle Arten von Daten, mit denen die Server-Komponente arbeitet, sowie die abstrakte Basisklasse für Datenbankadapter und die konkrete Implementierung dieser Basisklasse für MySQL-Datenbanken. Zusätzlich enthält der Namespace abstrakte und konkrete Fabrikklassen für Instanzen von Datenhaltungsklassen.

5.1.1.5 Orchid.Core.Exceptions

Speziell auf die Anforderungen der Server-Komponente zugeschnittene, selbst implementierte `Exception`-Klassen (Fehler-Klassen) sind in diesem Namespace untergebracht. Er ist in Abbildung 11 nicht aufgeführt.

5.1.1.6 Orchid.Core.Parsers

`Orchid.Core.Parsers` enthält diejenigen Klassen, die dafür zuständig sind, Anfragen, die eine Server-Instanz an eine Daemon-Instanz weiterreicht, zu analysieren und aufzubereiten (zu parsen).

5.1.1.7 Orchid.Core.Requests

In diesem Namespace befinden sich Klassen zur Kapselung eingehender Anfragen (Requests).

5.1.1.8 Orchid.Core.Servers

Der Namespace `Orchid.Core.Servers` enthält die abstrakten Basisklassen und konkreten Implementierungen für die zwei Arten von Server-Instanzen, die innerhalb eines ORCHID Servers existieren. Zusätzlich sind auch abstrakte und konkrete Fabrikklassen für Instanzen dieser Server in diesem Namespace enthalten.

5.1.1.9 Orchid.Core.Typeloading

Dieser Namespace enthält alle Klassen, die für das dynamische Typeloading (vgl. 6.3.2) notwendig sind.

5.1.2 Projektstruktur Client-Bibliothek

Die Client-Bibliothek des ORCHID Toolkits besteht aus drei Teilen. Kern ist die eigentliche Bibliothek namens `Orchid`, die wiederum die beiden anderen Teile, `OrchidInstruction` und `Strings` verwendet, wie in der folgenden Abbildung 12 dargestellt ist.

Abbildung 12 - Projektstruktur der Client-Bibliothek

Die einzelnen Bestandteile der Client-Bibliothek werden im Folgenden erläutert.

5.1.2.1 Orchid

Hierbei handelt es sich um die eigentliche Bibliothek, die es einem Client (Arduino) ermöglicht, mit einem ORCHID Server zu kommunizieren. Sie stellt alle dazu notwendigen Funktionen bereit und kann in eigenen Sketches verwendet werden.

5.1.2.2 OrchidInstruction

Um auch in der Client-Bibliothek einen möglichst objektorientierten Ansatz zu verwenden, wurden Instruktionen, die ein Server an einen Client senden kann, in eine eigene Klasse gekapselt. `OrchidInstruction` ist eine Datenhaltungsklasse, deren Instanzen jeweils eine Instruktion aufnehmen können.

5.1.2.3 Strings

Die Verarbeitung von Zeichenketten in einer Bibliothek für einen Arduino ist im Vergleich zu anderen Programmiersprachen aufwändiger. Entsprechende Funktionen wurden aus Gründen der leichteren Wartbarkeit und der Erhaltung der Übersichtlichkeit in die Bibliothek `Strings` ausgelagert.

5.2 Hierarchie in der Server-Komponente

Die Server-Komponente des ORCHID Toolkits verfügt über eine besondere Hierarchie, die an dieser Stelle verdeutlicht werden soll.

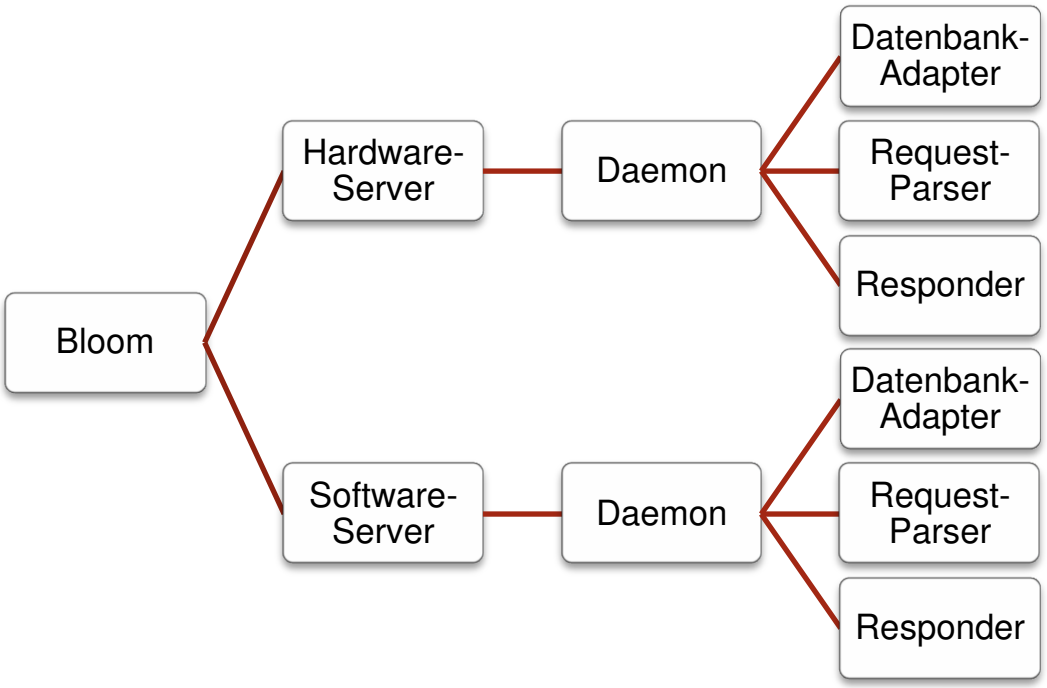

Abbildung 13 - Hierarchie in der Server-Komponente

Die obige Abbildung 13 zeigt die Klassen-Hierarchie, die in der Server-Komponente vorherrscht.

Eine *„Bloom“* (dt.: *„Blüte“*, eine Instanz der Klasse `Orchid.Core.Bloom`) verfügt über zwei separate Server (einen Hardware-Server und einen Software-Server). Diese Server warten auf eingehende Verbindungen an je einem Port. Der Hardware-Server nimmt Client-Verbindungen an, während der Software-Server Verbindungen von Drittsoftware entgegennimmt. `Bloom` bildet damit die oberste Ebene der dargestellten Hierarchie.

Jede Instanz eines Servers ist in der Lage, Instanzen von Daemons zu erzeugen. Die Art des Daemons richtet sich dabei nach der Art des Servers: ein Hardware-Server erzeugt Hardware-Daemons, ein Software-Server entsprechend Software-Daemons.

Da beide Arten von Daemons von der gleichen abstrakten Basisklasse (`AHardwareDaemon`) abgeleitet sind (`ASoftwareDaemon`, die abstrakte Basisklasse für Software-Daemons, ist von `AHardwareDaemon` abgeleitet), verhalten sie sich ähnlich. Beide Arten verwenden einen Datenbank-Adapter, um Daten zu speichern und zu lesen. Ein Request-Parser wird eingesetzt, um eingehende Anfragen von Clients bzw. Drittsoftware zu verarbeiten, eine Responder-Instanz wird zur Beantwortung eingehender Anfragen verwendet.

Um die Server-Komponente des ORCHID Toolkits in einer eigenen Implementierung eines ORCHID Servers zu nutzen, muss lediglich eine Instanz der Klasse `Bloom` erzeugt, initialisiert und deren Methode `Run()` aufgerufen werden.

5.3 Entwurfsmuster

Im Zuge der Implementierung der Server-Komponente kommen verschiedene Entwurfsmuster zum Einsatz. Diese werden im Folgenden genannt und anhand der konkreten Stellen, an denen sie im Code verwendet werden, erläutert.

5.3.1 Das Entwurfsmuster „Abstrakte Fabrik“

Da das im Rahmen dieser Arbeit entwickelte Toolkit insbesondere darauf ausgelegt ist, dass es von Dritten erweitert und verändert werden können soll, ist es notwendig, dass konkrete Klassenimplementierungen an möglichst wenigen Stellen im Code referenziert werden. Im Idealfall werden nur abstrakte Klassen eingesetzt, wodurch die konkreten Implementierungen jederzeit ausgetauscht werden können, ohne die sie verwendenden Klassen verändern zu müssen.

Um diese Vorgabe zu erfüllen, muss der Erzeugungsprozess von Objekten abstrahiert werden, so dass die Konstruktoren der zu erzeugenden Klassen nicht mehr von den sie aufrufenden Klassen verwendet werden – nur so kann später eine konkrete Klasse gegen eine andere ausgetauscht werden, ohne auch die aufrufende Klasse verändern zu müssen. Zu diesem Zweck wird das Entwurfsmuster *"Abstrakte Fabrik"* (engl. *"Abstract Factory Pattern"*) eingesetzt. Es abstrahiert den Erzeugungsprozess von Objekten, indem diese nicht mehr direkt erzeugt werden können, sondern dafür eine dritte Klasse, die sog. *"Fabrik"* (engl. *"Factory"*) zum Einsatz kommen muss. Eine Fabrik ermöglicht es einem Objekt, andere Objekte zu erzeugen, ohne die konkrete Implementierung eines solchen Objekts kennen zu müssen. Der Erzeuger des Objekts kennt nur eine abstrakte Basisklasse des zu erzeugenden Objekts (oder eine Schnittstelle) und arbeitet auch nur mit deren Methoden.

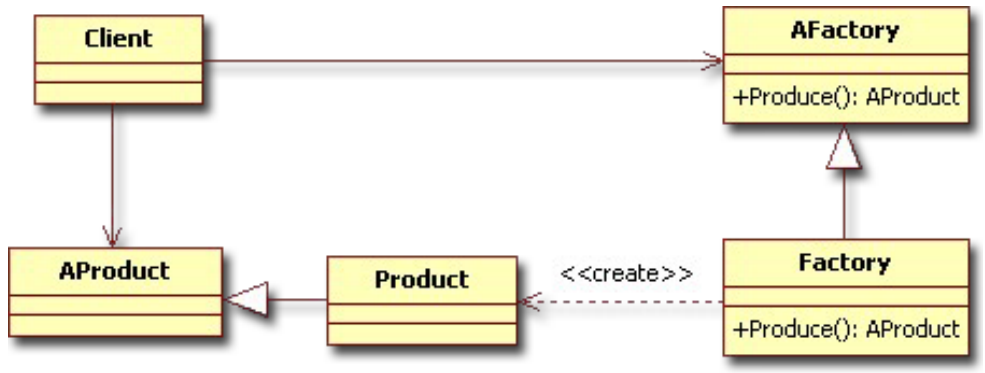

Abbildung 14 - Entwurfsmuster "Abstrakte Fabrik"

Die Abstrakte Fabrik unterscheidet verschiedene Rollen, die in Abbildung 14 zu sehen sind. Klienten (`Client`) können eine Fabrik (`Factory`) verwenden, um Objekte (`Product`) zu erzeugen. Der Klient kennt dabei nur die abstrakte Basisklasse (`AProduct`) und die abstrakte Fabrik (`AFactory`) und kommt mit den konkreten Implementierungen (`Product` und `Factory`) nicht in Berührung. Die Fabrik erzeugt Objekte über eine sog. "*Fabrikmethode*" (`Produce()`), indem sie die Konstruktoren der konkreten Klasse, die instanziiert werden soll, aufruft. Die konkrete Fabrik ist also die einzige Klasse, die den Konstruktor der konkreten Produktklasse nutzt.

Im Toolkit wird das Abstract Factory Pattern für alle zentralen Klassen der Server-Komponente verwendet. Für jede dieser Klassen existiert eine eigene Fabrikmethode; mehrere zusammengehörende Fabrikmethoden sind dabei in einer Fabrikklasse zusammengefasst.

Ausgehend vom zentralen Einstiegspunkt der Server-Komponente (Klasse `Bloom`, vgl. 5.1.1) werden alle benötigten Objekte, von den Server-Instanzen, über `Parser`, bis hin zu Datenhaltungsklassen (z.B. `Arduino`, `Sensor` oder `Actor`) ausschließlich über Fabriken erzeugt. Diese Vorgehensweise

erlaubt es, das Toolkit serverseitig mit eigenen Klassen zu erweitern, ohne bestehende Klassen verändern zu müssen.

Die folgende Abbildung zeigt die Verwendung des Entwurfsmusters innerhalb der Server-Komponente (*Server*- und *Daemon*-Klassen):

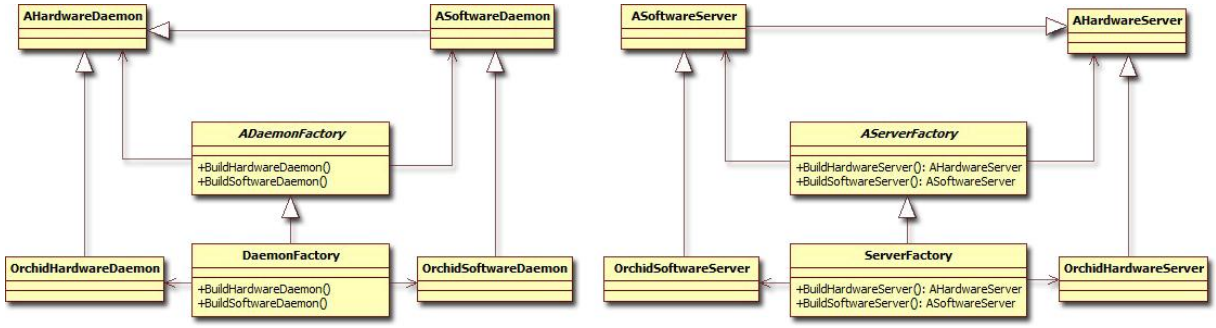

Abbildung 15 - Fabriken für Server und Daemons

Wie Abbildung 15 darstellt, gibt es zur Erzeugung von *Server*- und *Daemon*-Instanzen jeweils eine Fabrikklasse (*AServerFactory/ServerFactory* und *ADaemonFactory/DaemonFactory*). Jede dieser Fabrikklassen kann zwei verschiedene Klassen instanziieren (*HardwareServer* und *SoftwareServer* bzw. *HardwareDaemon* und *SoftwareDaemon*). Diese Doppelung wurde bewusst implementiert, da die Klassen, die durch die gezeigten Fabriken erzeugt werden, semantisch eng miteinander verbunden sind; eine Aufteilung in insgesamt vier Fabrikklassen erschien nicht sinnvoll.

5.3.1.1 Erweiterbarkeitsvorteile

Verwendet ein Entwickler die Server-Komponente und möchte eine konkrete Klasse selbst neu implementieren, ist dies ohne die Veränderung der bisherigen Klassen möglich. An einem Beispiel soll dies im Folgenden verdeutlicht werden.

```
01 Bloom bloom = new Bloom(new ServerFactory(), new DaemonFactory());
02 bloom.Blossom();
```

Listing 7 - Erzeugung eines Standard-ORCHID Servers

Listing 7 zeigt die Erzeugung eines ORCHID Servers entsprechend der Referenzimplementierung. Die Klasse *Bloom* verfügt über einen einzigen Konstruktor, der jeweils eine Instanz der abstrakten Klassen *AServerFactory* und *ADaemonFactory* als Parameter erwartet. Diese beiden Instanzen werden von *Bloom* dazu verwendet, die intern genutzten *Server*- und *Daemon*-Klassen zu erzeugen.

Verfügt ein Entwickler über eine eigene Implementierung von *AHardwareDaemon*, die als neue Daemon-Klasse in einem eigenen ORCHID-Server verwendet werden soll, muss lediglich eine entsprechende Fabrik erzeugt werden[38].

Dazu kann er seine Fabrikklasse von der abstrakten Klasse *ADaemonFactory* erben lassen und muss nun nur noch die Daemon-Erzeugung in dieser Klasse implementieren. Angenommen, die neuen Daemon-Klassen hießen *„CustomHardwareDaemon"* und *„CustomSoftwareDaemon"*, die neue Fabrikklasse *„CustomDaemonFactory"*. Dann würde sich eine Fabrikklasse wie in Listing 8 ergeben (vereinfachte Darstellung ohne Initialisierung der einzelnen Instanzen).

[38] Im regulären Abstract Factory Pattern. In der Server-Komponente kommen weitere Techniken zum Einsatz, die die Implementierung einer Fabrik optional machen (vgl. 6.3.2).

```
01 public class CustomDaemonFactory : ADaemonFactory
02 {
03      public override AHardwareDaemon BuildHardwareDaemon()
04      {
05          return new CustomHardwareDaemon();
06      }
07
08      public override ASoftwareDaemon BuildSoftwareDaemon()
09      {
10          return new CustomSoftwareDaemon();
11      }
12  }
```

Listing 8 - Eine neue Daemon-Fabrik

Der Aufruf der *Bloom*-Klasse verändert sich entsprechend, wie im folgenden Listing 9 zu sehen ist:

```
01 Bloom bloom = new Bloom(new ServerFactory(), new CustomDaemonFactory());
02 bloom.Blossom();
```

Listing 9 - Veränderte Erzeugung eines ORCHID Servers

Weitere Veränderungen am Code sind nicht notwendig. Die neuen Klassen *CustomHardwareDaemon* und *CustomSoftwareDaemon* werden automatisch innerhalb der Server-Instanzen verwendet.

5.3.2 Das Singleton-Entwurfsmuster

Das *"Singleton"*-Entwurfsmuster (*engl.: „singleton", dt.: „einelementige Menge"*) wird verwendet, wenn zur Laufzeit einer Software garantiert werden soll, dass es von einer bestimmten Klasse zu jedem Zeitpunkt der Ausführung nur eine Instanz gibt und dass es sich dabei stets um dieselbe Instanz handelt.

Das Singleton-Entwurfsmuster kommt innerhalb der Server-Komponente u.a. zum Einsatz, um die Klasse *Configuration* zu realisieren. Diese Klasse wird eingesetzt, um zahlreiche Einstellungen für die Server-Komponente des Toolkits zu verwalten und bereitzustellen. Die besondere Natur des Toolkits macht es dabei notwendig, dass eine Singleton-Klasse *threadsicher* ist, d.h. dass sie auch von mehreren Threads nebenläufig verwendet werden kann, ohne dabei die Singleton-Eigenschaften zu verlieren (vgl. 5.9.1).

Besonders deutlich wird diese Anforderung dann, wenn eine Singleton-Instanz zur Laufzeit erzeugt wird. Diese Erzeugung geschieht nur ein einziges Mal, nämlich beim ersten Zugriff auf die Instanz. Wird die Threadsicherheit nicht beachtet, kann eine Singleton-Klasse z.B. wie im nachfolgenden Listing 10 aufgebaut werden.

```
01 public sealed class NonThreadSafeSingleton
02 {
03      // Only instance is private
04      private static NonThreadSafeSingleton _instance;
05
06      // Prevent instances from being created
07      private NonThreadSafeSingleton() {}
08
09      // Instance property will create an instance if necessary
10      public static NonThreadSafeSingleton Instance
11      {
12           get
13           {
14               if(_instance == null)
15               {
16                    _instance = new NonThreadSafeSingleton();
17               }
18
19               return _instance;
20           }
21      }
22 }
```

Listing 10 - Nicht threadsichere Singleton-Klasse

Die im obigen Listing gezeigte Klasse `NonThreadSafeSingleton` ist eine Singleton-Klasse; sie kann nicht von außen instanziiert werden. Zugriffe auf die Klasse geschehen über die Property `Instance`, die – falls es zum Zeitpunkt eines Zugriffs noch keine Instanz von `NonThreadSafeSingleton` gibt – die einzige Instanz erzeugt.

In Umgebungen, in denen eine Software nur auf einem Thread arbeitet, kann diese Klasse gefahrlos eingesetzt werden. Sobald aber mehr als ein Thread existiert, kann es zu Problemen kommen. Es kann nicht vorausgesagt werden, wann welcher Thread auf welchen Codeabschnitt eines Programms zugreifen wird[39]. Entsprechend kann es vorkommen, dass aus einem Thread A heraus auf `Instance` zugegriffen wird, der den Code bis unmittelbar vor der Erzeugung der einzigen Instanz (`_instance`) abarbeitet und dann die Kontrolle wieder abgibt. Zu diesem Zeitpunkt existiert noch keine Instanz von `NonThreadSafeSingleton`. Nun greift Thread B auf `Instance` zu und arbeitet den Code bis zur Rückgabe ab. In diesem Moment existiert bereits eine Instanz von `NonThreadSafeSingleton`. Nachdem Thread B die Kontrolle wieder abgegeben hat, wird Thread A weiter ausgeführt. Die Prüfung, ob `_instance` verschieden von `null` ist, wurde aus Sicht von Thread A bereits abgearbeitet, eine neue Instanz soll erzeugt werden. Innerhalb von Thread A wird also eine neue Instanz der Klasse `NonThreadSafeSingleton` erzeugt und zurückgegeben. Es könnten somit zu einem Zeitpunkt zwei verschiedene Instanzen von `NonThreadSafeSingleton` entstehen, obwohl das Singleton-Entwurfsmuster garantieren soll, dass es zu jedem Zeitpunkt stets nur eine Instanz gibt. Dies kann zu Problemen im Programmablauf – bis hin zum Programmabsturz – führen, insbesondere wenn die Erzeugung von Instanzen mit einer komplexen Initialisierung verbunden ist.

Um eine Singleton-Klasse threadsicher zu machen, muss ein sog. *Mutex-Objekt* eingeführt werden. *Mutex* steht für *„mutual exclusion“* (*dt.: „wechselseitiger Ausschluss“*). Ein solches Objekt wird von allen Instanzen einer Klasse geteilt (`static`) und kann dazu verwendet werden, das o.g. Szenario zu vermeiden.

[39] vgl. http://msdn.microsoft.com/en-US/library/ms173178(v=VS.80).aspx

In der verwendeten Programmiersprache C# gibt es zu diesem Zweck den `lock`-Befehl, mit dem ein Objekt als Mutex verwendet werden kann. Die in Listing 10 gezeigte, nicht threadsichere Klasse kann mit diesem Verfahren threadsicher gemacht werden, wie das folgende Listing 11 zeigt.

```
01 public sealed class ThreadSafeSingleton
02 {
03     // Only instance is private
04     private static ThreadSafeSingleton _instance;
05
06     // Lock object
07     private static object _lockObject = new Object();
08
09     // Prevent instances from being created
10     private ThreadSafeSingleton() {}
11
12     // Instance Property will create an instance if necessary
13     public static ThreadSafeSingleton Instance
14     {
15         get
16         {
17             if(_instance == null)
18             {
19                 lock(_lockObject)
20                 {
21                     if(_instance == null)
22                     {
23                         _instance = new ThreadSafeSingleton();
24                     }
25                 }
26             }
27
28             return _instance;
29         }
30     }
31 }
```

Listing 11 - Threadsichere Singleton-Klasse

Trifft in der Klasse *ThreadSafeSingleton* ein Thread auf die Erzeugungsroutine der einzigen Instanz *_instance*, wird wie zuvor geprüft, ob es bereits eine solche Instanz gibt. Ist dies nicht der Fall, wird das Mutex-Objekt (*_lockObject*) über den `lock`-Befehl gesperrt. Kein anderer Thread kann nun in den damit eingeschlossenen Codeabschnitt hineinlaufen, die Erzeugung wird garantiert nur von einem Thread übernommen. Trifft ein anderer Thread auf die Stelle im Code, an der das Mutex-Objekt gesperrt wird, und ist dieses Objekt bereits gesperrt, wird die Ausführung des Threads an dieser Stelle unterbrochen und der Thread gibt die Kontrolle wieder ab. Er kann also unmöglich eine zweite Instanz von *ThreadSafeSingleton* erzeugen.

5.3.3 Dependency Injection

Dependency Injection (dt. etwa: „Abhängigkeits-Injektion“) beschreibt ein Entwurfsmuster für objektorientierte Systeme. Es handelt sich nicht um ein klassisches Entwurfsmuster (vgl. [23] und [24]), sondern vielmehr um eine Technik zur Reduzierung von starren Abhängigkeiten innerhalb eines Programms, die sich zur Abhängigkeitsreduzierung des Prinzips der *Inversion of Control (dt.: „Kontroll-Umkehrung“)* bedient.

In der sequentiellen Programmierung existiert im Code ein Hauptabschnitt, der nach und nach Methoden aufruft, um bestimmte Aufgaben durchzuführen. Ist eine Methode abgearbeitet, wird der

Programmfluss wieder auf den Hauptabschnitt umgeleitet, der die nächste Methode aufruft. Diese Schritte werden wiederholt, bis die letzte Zeile des Hauptabschnitts erreicht ist, woraufhin das Programm beendet wird.

Inversion of Control bedeutet, dass dieser Prozess umgekehrt wird. Der Hauptabschnitt kann einzelne Methoden bei einer zentralen Stelle (z.B. einem Framework) registrieren. Dazu muss angegeben werden, unter welchen Bedingungen diese Methoden ausgeführt werden sollen (also z.B. wenn ein bestimmtes Ereignis eintritt). Der Hauptabschnitt ruft die Methoden nicht mehr selbst auf, sondern meldet nur noch eintretende Ereignisse an die zentrale Stelle, die daraufhin die passenden Methoden aufruft. Ein gutes Beispiel für die Verwendung von Inversion of Control sind die sog. *„Callback-Methoden (dt. etwa: „Rückruf-Methoden“)* der Programmierung grafischer Benutzeroberflächen. Eine Callback-Methode (auch *„Callback“* genannt) wird einem Ereignis eines Steuerelements auf der Benutzeroberfläche zugewiesen (z.B. dem Anklicken eines bestimmten Buttons), also bei einer zentralen Stelle registriert. Wann immer das Ereignis des Steuerelements eintritt, wird die registrierte Callback-Methode durch die zentrale Stelle aufgerufen, der eigentliche Hauptabschnitt des Codes ist dafür nicht länger zuständig.

Das Prinzip der Inversion of Control wird im Dependency Injection Entwurfsmuster lediglich zur Erzeugung von Objekten und deren Abhängigkeiten verwendet. Dependency Injection kann also als weitere Abstrahierung des Abstract Factory Pattern gesehen werden, das bereits in 5.3.1 beschrieben wurde.

In der klassischen objektorientierten Programmierung ist jedes Objekt selbst dafür zuständig, seine Abhängigkeiten zu initialisieren, sei es über Konstruktor-Parameter, Initialisierungsmethoden oder andere Vorgehensweisen (oft als *„manuelle Dependency Injection“* bezeichnet). Dependency Injection greift maßgeblich in diesen Prozess ein, indem es nicht nur die Erzeugung von einzelnen Objekten, sondern auch die Initialisierung ihrer Abhängigkeiten externalisiert. Anstatt den Erzeugungscode für ein bestimmtes Objekt immer dann auszuführen, wenn eine neue Instanz eines solchen Objekts benötigt wird, wird bei einer zentralen Stelle (Dependency Injection Framework) registriert, wie das Objekt zu erzeugen ist. Wenn nötig, können neue Instanzen vom Programmcode bei der zentralen Stelle angefordert werden.

Dieser Abschnitt befasst sich ausschließlich mit Dependency Injection im Allgemeinen. Die konkrete Verwendung dieses Entwurfsmusters in der Server-Komponente des ORCHID Toolkits wird in Abschnitt 6.3.2 detailliert erläutert.

5.3.3.1 Vorteile

Die Vorteile der Dependency Injection liegen insbesondere in der Reduzierung von Abhängigkeiten innerhalb des Codes. Ähnlich wie im Abstract Factory Pattern ist es bei Verwendung dieses Musters nicht notwendig, dass Objekte die konkreten Implementierungen der von ihnen verwendeten Abhängigkeiten kennen; eine Schnittstelle oder abstrakte Basisklasse genügt. Durch diese Vorgehensweise werden die Abhängigkeiten zwischen einzelnen Klassen reduziert, konkrete Implementierungen können leicht gegeneinander ausgetauscht werden, ohne die Implementierung anderer Klassen verändern zu müssen.

Ein weiterer Vorteil, der sich insbesondere bei sehr komplexen Objekten zeigt, entsteht ebenfalls durch die Abhängigkeitsreduzierung. Wird in der traditionellen objektorientierten Programmierung ein sehr komplexes Objekt erzeugt, das wiederum viele Objekte vieler unterschiedlicher anderer Klassen verwendet, müssen alle verwendeten Klassen bekannt sein und die entsprechenden Objekte durch das komplexe Objekt initialisiert werden bzw. bei dessen Erzeugung bereits initialisiert sein. Der Aufwand für einen Entwickler ist entsprechend hoch, die Lesbarkeit und Wartbarkeit des Codes wird zum Teil

stark beeinträchtigt (je nach Komplexität der betroffenen Objekte). Wird stattdessen das Dependency Injection Entwurfsmuster verwendet, können sowohl die Codekomplexität verringert, als auch die Lesbarkeit und Wartbarkeit des Codes erheblich erhöht werden.

Durch die äußerst geringe Kopplung zwischen einzelnen Klassen, die sich durch die Verwendung von Dependency Injection erreichen lässt, ergibt sich auch der Vorteil einer hohen Flexibilität. Je nach eingesetztem Framework ist es so z.B. möglich, die zu verwendenden konkreten Klassen auszutauschen, ohne den Code einer Anwendung neu übersetzen zu müssen. Dazu ist jedoch ein erhöhter Konfigurationsaufwand notwendig, wie in 6.3.2 im Detail erläutert wird. Da durch die zusätzliche Flexibilität, die Dependency Injection einem Projekt verleiht, jedoch dessen Erweiterbarkeit sehr stark erhöht wird, überwiegt der Vorteil.

5.3.3.2 Nachteile

Dependency Injection erfordert aufwändige Prozesse, um Objekte zu erzeugen. Diese Prozesse werden meist in einem Framework gekapselt, das von Entwicklern genutzt wird, um das Entwurfsmuster zu verwenden. Damit das Entwurfsmuster in möglichst jedem Szenario funktionieren kann, müssen solche Frameworks häufig über externe Dateien (oder direkt aus dem Code heraus) konfiguriert werden. Diese Konfiguration ist oft sehr aufwändig und verringert die Lesbarkeit des Codes, da zum Verständnis betroffener Codeabschnitte auch die Konfiguration der Software bzw. des Frameworks betrachtet werden muss.

Je nach Dependency Injection Framework und den darin verwendeten Techniken kann es auch zu Performance-Einbußen kommen. Dies gilt insbesondere für die Verwendung von *Reflection* (unter *„Reflection“* werden Techniken zusammengefasst, die es einem Programm erlauben, sich selbst zur Laufzeit zu analysieren und zu verändern).

5.3.3.3 Beispiel

Ohne auf ein konkretes Dependency Injection Framework einzugehen, sollen in diesem Abschnitt die Verwendung des Musters und dessen Vorteile erläutert werden. Die einzige Annahme, die über das verwendete Framework getroffen wird, ist dessen Konfigurierbarkeit über eine externe Datei.

Es wird angenommen, dass ein abstraktes Softwareprojekt unter Verwendung von Dependency Injection entwickelt wurde. In diesem Softwareprojekt gibt es eine zentrale Klasse `Center`, die die hauptsächliche Arbeit (deren Art an dieser Stelle nicht von Bedeutung ist) der Anwendung leistet. Die Hauptaufgabe der Anwendung soll nun verändert werden, allerdings nicht vom ursprünglichen Entwickler, sondern von einem Dritten. Dieser Dritte kennt die abstrakte Basisklasse `ACenter`, die als Basis für die Klasse `Center` dient. Er schreibt nun eine eigene Implementierung dieser Klasse namens `Center2`, die an die neuen Anforderungen angepasst ist.

Um die Anwendung so zu verändern, dass sie nicht mehr die alte, sondern die neue Aufgabe erledigt, muss der Entwickler nur die Konfigurationsdatei so anpassen, dass anstatt `Center` nun die neu entwickelte Klasse `Center2` verwendet wird. Die Bereitstellung dieser Klasse ist abhängig vom verwendeten Dependency Injection Framework und wird an dieser Stelle nicht behandelt.

Ist die Konfigurationsdatei angepasst und wird die Anwendung gestartet, wird nun an jeder Stelle, an der zuvor eine Instanz von `Center` erzeugt wurde, eine Instanz von `Center2` erzeugt, die die neuen Anforderungen abdeckt (in dem Rahmen, den die abstrakte Basisklasse `ACenter` vorgibt). Die Anwendung wurde also stark verändert, ohne sie im Kern neu kompilieren oder auch nur eine einzige Zeile Quellcode verändern zu müssen.

5.4 Verwendete Technologien

Dieser Abschnitt befasst sich mit den für die Implementierung der Toolkit-Komponenten verwendeten Technologien, geht auf deren für die vorliegende Arbeit interessanten Besonderheiten ein und hebt besondere Vorteile, die eine Technologie für die Entwicklung von Software bietet, hervor.

5.4.1 C# und das Mono Framework

Einige der sich aus der Verwendung des bereits in 2.3.1 beschriebenen Mono Frameworks ergebenden programmiertechnischen Vorteile werden im Folgenden erläutert. Die hier beschriebenen Möglichkeiten, die Mono einem Entwickler bietet, wurden speziell im Hinblick auf mögliche Erweiterungen oder Veränderungen der Server-Komponente des ORCHID Toolkits ausgewählt.

Alle Code-Beispiele und Technologiebeschreibungen in diesem Abschnitt beziehen sich auf die Programmiersprache C# und das Mono Framework.

5.4.1.1 Properties

In der objektorientierten Programmierung ist es üblich, dass Instanzen von Klassen die in ihnen enthaltenen Variablen nach außen verbergen, so dass andere Instanzen nicht darauf zugreifen können (`private`). Soll der Zugriff möglich sein, werden der Klasse entsprechende Methoden hinzugefügt, die eine private Variable auslesen oder verändern.

Ein Beispiel für diese Vorgehensweise kann dem folgenden Listing entnommen werden.

```
01 public class GetterAndSetter
02 {
03     private int _theAnswer = 42;
04
05     public int GetTheAnswer()
06     {
07         return this._theAnswer;
08     }
09
10     public void SetTheAnswer(int newAnswer)
11     {
12         this._theAnswer = newAnswer;
13     }
14 }
```

Listing 12 - Beispielklasse ohne Properties

Werden Instanzen der im obigen Listing gezeigten Klasse `GetterAndSetter` verwendet, kann dies durch die Notwendigkeit der Zugriffsmethoden `GetTheAnswer()` und `SetTheAnswer(int)` zu Code führen, der schlecht lesbar ist, wie das folgende Listing beweist.

```
01 private void TestGetterAndSetter(GetterAndSetter a, GetterAndSetter b)
02 {
03     a.SetTheAnswer(b.GetTheAnswer() * a.GetTheAnswer());
04 }
```

Listing 13 - Anwendung von Zugriffsmethoden

Der Code in Listing 13 ist unvorteilhaft, da der einzige Unterschied zwischen den Zugriffsmethoden der erste Buchstabe ihres Namens ist. So kann es leicht zu Verwechselungen der Methoden kommen. Die zahlreichen Klammern, die bei der Verwendung von Zugriffsmethoden gesetzt werden müssen, verschlechtern die Lesbarkeit weiter und können zu Fehlern in komplexen Berechnungen oder logischen Vergleichen führen.

C# bietet mit *Properties (dt.: „Eigenschaften“)* ein Sprachkonstrukt an, das dieses Problem beseitigt und gleichzeitig die Anzahl der für die Implementierung einer Klasse notwendigen Zeilen stark verringert. Das folgende Listing zeigt die in Listing 12 mit Zugriffsmethoden implementierte Klasse `GetterAndSetter` unter Verwendung von Properties.

```
01 public class GetterAndSetter
02 {
03     private int _theAnswer = 42;
04
05     public int TheAnswer
06     {
07         get { return this._theAnswer; }
08         set { this._theAnswer = value; }
09     }
10 }
```

Listing 14 - Beispielklasse mit Properties

In ihrer Funktionalität entspricht die in Listing 14 gezeigte Klasse exakt der in Listing 12 gezeigten, kommt aber mit weniger Codezeilen aus. Auch die Verwendung im Code, die in Listing 15 gezeigt wird, ist übersichtlicher und intuitiver als die in Listing 13 demonstrierte.

```
01 private void TestGetterAndSetter(GetterAndSetter a, GetterAndSetter b)
02 {
03     a.TheAnswer = b.TheAnswer * a.TheAnwser;
04 }
```

Listing 15 - Anwendung von Properties

Obwohl das obige Beispiel genau die gleiche Operation durchführt wie das zuvor beschriebene, ist der Code weitaus leichter lesbar. Fehler durch falsche Klammerung sind weniger wahrscheinlich, da durch die Verwendung von Properties die Menge an notwendigen Klammern erheblich reduziert wird. Darüber hinaus kann die Operation – im Gegensatz zum in Listing 13 gezeigten Code - sofort als Zuweisung interpretiert werden.

Properties erlauben es zusätzlich, unterschiedliche Modifizierer für den Zugriff zu verwenden, die im folgenden Listing dargestellt werden.

```
01 public class GetterAndSetter
02 {
03     private int _theAnswer = 42;
04
05     public int TheAnswer
06     {
07         get { return this._theAnswer; }
08         protected internal set { this._theAnswer = value; }
09     }
10 }
```

Listing 16 - Beispielklasse mit unterschiedlichen Zugriffsmodifizierern

Der lesende Zugriff auf die private Variable `_theAnswer` ist nach wie vor öffentlich, schreibende Zugriffe werden nur Klassen erlaubt, die von der Klasse `GetterAndSetter` abgeleitet werden oder sich in derselben Assembly befinden (`protected internal`).

5.4.1.2 Extension Methods

Eine *Extension Method* (*dt.: „Erweiterungsmethode“*) erlaubt es einem Entwickler, einer bestehenden Klasse beliebige Funktionen hinzuzufügen, ohne die Klasse selbst verändern zu müssen.

Extension Methods werden als statische Methoden in einer statischen Klasse definiert, aber wie Instanzmethoden der Objekte der von ihnen erweiterten Klasse aufgerufen. Obwohl die ursprüngliche Klasse also nicht verändert wurde, bieten ihre Instanzen nach der Erweiterung neue Methoden an, genau so, als wären diese Methoden bereits als Instanzmethoden der ursprünglichen Klasse definiert worden.

Insbesondere im Hinblick auf die nachträgliche Erweiterung und Anpassung der Server-Komponente sind Extension Methods sehr nützlich. Möchte ein Entwickler einer bestehenden Klasse eine Methode hinzufügen, kann er dies tun, ohne den ursprünglichen Code zu verändern.

Das folgende Listing zeigt ein Beispiel für eine Extension Method, die die Klasse `AHardwareServer`, also die grundlegende Klasse für alle ORCHID Server, erweitert.

```
01 public static class ServerExtension
02 {
03     public static TimeSpan GetCurrentRuntime(this AHardwareServer server)
04     {
05         return DateTime.Now.Subtract(server.Started);
06     }
07 }
```

Listing 17 - Beispielhafte Extension-Method

Die oben gezeigte Methode `GetCurrentRuntime` erwartet einen Parameter vom Typ `AHardwareServer`. Das Schlüsselwort `this` ist hier notwendig, da es sich um eine Extension Method handelt. Fehlt es, wird die Methode als normale statische Methode betrachtet und kann nicht über Instanzen von Klassen, die von `AHardwareServer` abgeleitet wurden, aufgerufen werden.

`GetCurrentRuntime(AHardwareServer)` berechnet die aktuelle Laufzeit eines Servers, indem dessen Startzeitpunkt von der aktuellen Zeit subtrahiert wird. Ein Aufruf der Methode im Code geschieht wie im folgenden Listing gezeigt.

```
01 private void ShowServerRuntime(AHardwareServer hardwareServer)
02 {
03     TimeSpan ts = hardwareServer.GetCurrentRuntime();
04     Console.Write("Hardware Server is running since ");
05     Console.Write(ts.TotalSeconds().ToString());
06     Console.WriteLine(" seconds");
07 }
```

Listing 18 - Anwendung einer Extension Method

Ohne die Klasse `AHardwareServer` oder eine davon abgeleitete Klasse verändert zu haben, kann nun im Code die aktuelle Laufzeit einer Server-Instanz ermittelt werden, als ob die entsprechende Methode eine Instanzmethode der Klasse selbst wäre.

5.4.1.3 Lambda Expressions

Das Mono Framework bietet die Möglichkeit, das Konstrukt der *Lambda Expressions (dt.: „Lambda-Ausdrücke“)* der Programmiersprache C# zu nutzen. Eine Lambda Expression ist eine anonyme Methode, die u.a. als Methodenparameter an andere Methoden übergeben werden kann. Durch diese

Eigenschaft ergeben sich zahlreiche interessante und vorteilhafte Möglichkeiten, Lambda Expressions für die Erweiterung der Server-Komponente einzusetzen.

Insbesondere in Verbindung mit den bereits beschriebenen Extension Methods sind Lambda Expressions ein mächtiges Werkzeug, um bestehenden Code zu erweitern, ohne ihn zu verändern. Das folgende Beispiel zeigt eine weitere Extension Method, die als Parameter eine Lambda Expression erwartet.

```
01 public static class ServerExtension
02 {
03     public static IEnumerable<ASensorData> Filter(this List<ASensorData> list,
                                                      Func<ASensorData, bool> p)
04     {
05         foreach(ASensorData data in list)
06         {
07             if(p(data)) { yield return data; }
08         }
09     }
10 }
```

Listing 19 - Extension Method mit Lambda Expression

Die hier gezeigte Extension Method `Filter` erwartet als Parameter eine Liste von Sensordaten (sie erweitert also die Klasse `List<ASensorData>`) und ein Prädikat (eine Lambda Expression). Sie geht alle Elemente der übergebenen Liste durch und prüft für jedes Element, ob das Prädikat für dieses Element den Wert `true` liefert. Ist dies der Fall, wird das Element zurückgegeben, ansonsten verworfen.

Um diese Extension Method im eigenen Code verwenden zu können, muss ein Entwickler die Lambda Expression angeben, die als Prädikat übergeben werden soll.

```
01 private IEnumerable<ASensorData> FilterSensorData(List<ASensorData> list)
02 {
03     return list.Filter(x => x.DataType == "INT");
04 }
```

Listing 20 - Exemplarische Lambda Expression

Im obigen Beispiel wird angenommen, dass der Methodenparameter `list` eine ungefilterte Liste von Sensordaten enthält. Ein Entwickler möchte eventuell aber nur Sensordaten erhalten, die Integer-Werte enthalten (gekennzeichnet durch das Kürzel *„INT"*). Mit der Lambda Expression `x => x.DataType == "INT"` werden genau diese Werte gefiltert. Die Expression wird gelesen als *„für x prüfe, ob x.DataType gleich 'INT' ist"* und liefert den Wahrheitswert dieser Aussage zurück.

In der Extension Method wird diese Lambda Expression als Prädikat auf jedes Element der übergebenen Liste angewendet (anstelle von `x` eingesetzt) und die Liste somit gefiltert.

Die hier beschriebene Funktionalität kann natürlich auch ohne Lambda Expressions implementiert werden; dabei geht jedoch ein Großteil der enormen Flexibilität verloren, die die gezeigte Vorgehensweise bietet. Möchte ein Entwickler zu einem späteren Zeitpunkt nicht mehr nach Integer-Werten filtern, muss keine neue Methode geschrieben, sondern lediglich die übergebene Lambda Expression verändert werden. Auch gänzlich andere Arten der Filterung, die wesentlich komplexer sein können, können über das hier gezeigte Verfahren angewendet werden, ohne die Filter-Methode verändern zu müssen.

5.4.1.4 LINQ

LINQ steht für *„Language Integrated Query“* (dt.: *„in die Sprache integrierte Abfrage“*) und erlaubt es, SQL-ähnliche Abfragen auf beliebige Objekte anzuwenden. Diese Abfragen werden dabei nicht als Zeichenkette übergeben, wie es z.B. bei Datenbanken der Fall ist, sondern bestehen aus einer Reihenfolge von Schlüsselwörtern und Ausdrücken, so dass sie bereits durch den Compiler geprüft werden können, was die Fehlersuche erheblich vereinfacht und Laufzeitfehler durch (syntaktisch) fehlerhafte Abfragen ausschließt.

LINQ bietet eine Vielzahl von Anwendungsmöglichkeiten, deren Auflistung den Rahmen dieser Arbeit sprengen würde. Stattdessen soll auf die Verwendung von LINQ als Abfragesprache für Listen eingegangen werden, da hier das volle Potential des Sprachkonstrukts deutlich erkennbar ist.

Eine typische Aufgabe in der Programmierung ist die Zusammenführung und Filterung zweier Listen von Elementen des gleichen Datentyps, die aus unterschiedlichen Quellen stammen, beispielsweise aus zwei gesonderten Datenbankabfragen. Ohne die Verwendung eines Konstrukts wie LINQ ist der Einsatz einer oder mehrerer Schleifen notwendig, wie im folgenden Listing zu sehen ist.

```
01 private List<int> MergeAndFilter(List<int> first, List<int> second)
02 {
03     List<int> result = new List<int>();
04
05     foreach (int element_first in first)
06     {
07         if(element_first % 2 == 0) { result.Add(element_first); }
08     }
09
10     foreach (int element_second in second)
11     {
12         if(element_second % 2 == 0) { result.Add(element_second); }
13     }
14
15     return result;
16 }
```

Listing 21 - MergeAndFilter mit Schleifen

Das obige Beispiel zeigt, wie unter Verwendung von zwei Schleifen zwei Listen von Integer-Werten zusammengeführt werden, wobei lediglich diejenigen Werte erhalten bleiben, die ohne Rest durch zwei teilbar sind (gerade Zahlen).

Die gleiche Funktionalität kann unter Verwendung von LINQ wie folgt implementiert werden:

```
01 private List<int> MergeAndFilter(List<int> first, List<int> second)
02 {
03    return first.Where<int>(x => x % 2 == 0)
            .Union<int>(second.Where<int>(y => y % 2 == 0)).ToList<int>();
04 }
```

Listing 22 - MergeAndFilter mit LINQ

Bei gleicher Eingabe liefern die in Listing 21 und Listing 22 gezeigten Methoden exakt das gleiche Ergebnis zurück. Die Redundanz im Code, die die Verwendung von zwei Schleifen im ersten Beispiel verursacht, wird durch den Einsatz von LINQ entfernt, die Lesbarkeit zusätzlich erhöht.

Aus dem obigen Beispiel in Listing 22 wird ersichtlich, dass LINQ zu großen Teilen als Extension Methods implementiert wurde (`Where<T>`, `Union<T>` und `ToList<T>` sind generische Extension

Methods der Klasse `List<T>`). Es existiert darüber hinaus jedoch noch eine zweite Form, die den aus SQL bekannten Abfragen noch stärker ähnelt als die bisher gezeigte.

Das folgende Beispiel zeigt die Filterung und aufsteigende Sortierung einer Liste von Ganzzahlen (nur gerade Zahlen sollen erhalten bleiben) unter Verwendung von LINQ.

```
01 private IEnumerable<int> FilterAndSort(List<int> list)
02 {
03     return from x in list where x % 2 == 0 orderby x ascending select x;
04 }
```

Listing 23 - FilterAndSort mit LINQ

Diese Methode liefert alle geraden Zahlen aus `list` aufsteigend sortiert zurück. Ohne die Verwendung von LINQ wären dazu mindestens eine Schleife und ein einfacher Sortieralgorithmus notwendig.

Die Möglichkeiten, die LINQ einem Entwickler bietet, sind natürlich nicht auf die hier gezeigten Anwendungsfälle begrenzt. Komplexe Abfragen können, ähnlich wie bei der Verwendung relationaler Datenbanken, Daten auf zahlreiche Arten filtern und aufbereiten.

5.4.2 Arduino und C/C++

Wie bereits in 2.1.2.4 beschrieben, erleichtert die Sprache Arduino die Programmierung eines Microcontrollers erheblich, ohne den Funktionsumfang einzuschränken. Dies ist möglich, da Arduino auf C/C++ aufbaut und Sketches zunächst in für diese Sprachen gültigen Code umgesetzt und erst danach kompiliert werden.

Aus der Basis von Arduino in den Sprachen C und C++ ergeben sich verschiedene Vor- und Nachteile, die im Folgenden beschrieben werden.

5.4.2.1 Geschwindigkeit

C und C++ sind sehr maschinennahe Programmiersprachen[40], die einem Entwickler die vollständige Kontrolle über die ausführende Hardware geben und damit deren volles Potential ausnutzen können, was zu erheblichen Geschwindigkeitsvorteilen führen kann. Diese Eigenschaft ist insbesondere deshalb ein Vorteil, da Microcontroller nicht über leistungsstarke Prozessoren verfügen, sondern im Vergleich sehr langsam arbeiten. Ein aktueller Prozessor für einen Privatcomputer arbeitet mit Taktfrequenzen im Gigahertz-Bereich, wohingegen Microcontroller des auf einem Arduino I/O-Board verbauten Typs mit 16 Megahertz arbeiten (vgl. 2.1.2.1). Im Vergleich zu einem Prozessor mit einer Taktfrequenz von 2 GHz arbeitet ein Arduino also 125 Mal langsamer.

Der Nachteil, der sich aus der Maschinennähe von C und C++ ergibt, ist der hohe Aufwand, der zur Programmierung in diesen Sprachen notwendig ist. Durch die Verwendung von Arduino zur Entwicklung von Sketches wird dieser Nachteil jedoch relativiert.

5.4.2.2 Kontrolle

C und C++ erlauben es einem Programmierer, die vollständige Kontrolle über das ausführende System zu übernehmen. So können Hardwarefunktionen direkt angesprochen und Speicherbereiche exakt manipuliert werden. Dies ist insbesondere für die Programmierung eines Microcontrollers wichtig, da es gerade die Funktionen der Hardware sind, die dabei von Interesse sind. Auch die exakte Manipulation des Speichers kann hierbei als Vorteil angesehen werden, da dieser begrenzt ist und effizient genutzt werden muss.

[40] „Maschinennähe“ beschreibt den Grad der Ähnlichkeit einer Programmiersprache mit Maschinensprache.

Die vollständige Kontrolle über ein System führt dazu, dass ein Programmierer stets in der Verantwortung ist, einmal reservierte Ressourcen, z.B. Bereiche im Hauptspeicher, nach deren Verwendung wieder freizugeben. Dies kann bei unsachgemäßer Anwendung dazu führen, dass Programme bei längerer Laufzeit instabil werden oder sogar abstürzen. Ein weiterer Nachteil, der sich ebenfalls aus dem hohen Maß an Kontrolle ergibt, ist die Gefahr, das gesamte ausführende System in einen instabilen Zustand zu versetzen. Dies kann u.a. bei schreibenden Speicherzugriffen geschehen, die auf einem nicht dafür vorgesehenen Speicherbereich ausgeführt werden.

5.4.3 Arduino Microcontroller-Plattform

Es gibt viele verschiedene Microcontroller auf dem Markt, deren Einsatz jeweils eigene Vor- und Nachteile mit sich bringt. Im Folgenden wird erläutert, warum die Wahl für diese Arbeit auf die Verwendung der Arduino Microcontroller-Plattform fiel.

Hard- und Software der Plattform sind quelloffen. Das bedeutet, dass neben dem Quellcode der Entwicklungsumgebung und des Compilers auch die für die Herstellung von Arduino Microcontroller-Boards notwendigen Schaltpläne frei verfügbar sind.

Durch diese Offenheit sind verschiedene Nachbauten entstanden, die die Einsatzmöglichkeiten des Originals zum Teil erheblich erweitern (z.B. das *„Boarduino“*[41]) und voll kompatibel zu regulären Arduino I/O-Boards sind.

Da die Hardware quelloffen ist, müssen Arduino I/O-Boards nicht zwingend vom Hersteller bezogen werden, da auch die eigene Herstellung mit im Elektronik-Fachhandel günstig zu beziehenden Komponenten möglich ist. Dies ist insbesondere dann von Vorteil, wenn eine Schaltung, die über einen Arduino Microcontroller gesteuert wird, fest verbaut werden soll, da in diesem Fall alle zum Betrieb notwendigen Komponenten direkt in die entsprechende Schaltung integriert werden können.

Die niedrige Einstiegshürde der Arduino-Plattform, die u.a. auf der vereinfachten Programmierung beruht, der geringe Kostenfaktor für die benötigte Hardware und die Verfügbarkeit der Entwicklungsumgebung für Windows, Linux und MacOS, macht die Arduino Microcontroller-Plattform sehr beliebt.

5.5 Frameworks und Tools

In diesem Abschnitt der Arbeit wird beschrieben, welche Frameworks und Tools bei der Entwicklung der Server-Komponente zum Einsatz kamen. Dieser Abschnitt konzentriert sich auf die Server-Komponente, da für die Entwicklung der Client-Bibliothek keine besonderen Frameworks oder Tools eingesetzt wurden.

5.5.1 NLog

Da Server-Software im Gegensatz zu Anwender-Software meist auf einem System ausgeführt wird, das ohne grafische Benutzeroberfläche auskommt und die Ausgaben einer solchen Software oft sogar gänzlich ausgeblendet werden (z.B. wenn die Software im Hintergrund läuft), ist es sehr wichtig, Meldungen in eine Log-Datei auszugeben. Für Tests oder die Entwicklung eigener Projekte mit dem ORCHID Toolkit muss aber auch eine Ausgabe der Log-Meldungen auf dem Bildschirm möglich sein.

Aus diesen Gründen musste ein flexibles Logging-Framework gefunden werden, dass die o.g. Anforderungen erfüllt, ohne zu sehr in den Programmablauf einzugreifen. Weiteres Kriterium bei der Suche nach einem Logging-Framework war der Konfigurationsaufwand, der zum Einsatz des Frameworks notwendig ist, sowie wiederum die Flexibilität der Konfiguration.

[41] http://www.ladyada.net/make/boarduino/

Die Wahl fiel schließlich auf *NLog*[42]. NLog ist kostenlos verfügbar und unter der BSD-Lizenz[43] veröffentlicht, weshalb die Nutzung innerhalb des ORCHID Toolkits keine lizenzrechtlichen Probleme bereitet. NLog bietet eine voll funktionsfähige Logging-Umgebung, die direkt auf die Verwendung in .NET/Mono zugeschnitten ist und sich entsprechend nahtlos in den eigenen Code einfügen lässt.

Der Konfigurationsaufwand, der zum Betrieb von NLog notwendig ist, ist im Vergleich zu anderen Logging-Frameworks (z.B. *log4net*[44]) etwa gleich, oft jedoch geringer und flexibler. Bei anderen Logging-Frameworks handelt es sich oft um Portierungen von Java-Frameworks (hier insbesondere log4net, wobei es sich um eine Portierung des log4j-Frameworks[45] handelt). NLog ist keine Portierung, sondern speziell auf die .NET/Mono-Umgebung ausgerichtet und von Grund auf dafür entwickelt; ein solches Framework ist bei gleichem Funktionsumfang einer Portierung vorzuziehen.

Ein weiterer Vorteil von NLog, den nicht alle verfügbaren Logging-Frameworks bieten, ist die Konfiguration aus dem eigenen Code während der Laufzeit. Die meisten Logging-Frameworks erfordern das Vorhandensein einer eigenen Konfigurationsdatei, die vom Framework eingelesen und zur Konfiguration verwendet wird. Die Konfiguration von NLog kann wahlweise über eine Konfigurationsdatei oder aus dem Code vorgenommen werden.

5.5.1.1 Flexible Konfiguration aus dem Quellcode

Da die Konfiguration von NLog aus dem Quellcode einer Anwendung heraus für die Verwendung und Anpassung der Server-Komponente von Bedeutung ist, soll sie an dieser Stelle erläutert werden.

Häufig werden externe Logging-Frameworks über eine oder mehrere Konfigurationsdateien konfiguriert, wobei es sich dabei meist um XML-Dateien handelt. Auch NLog kann über eine solche Datei konfiguriert werden.

```
01 <?xml version="1.0" encoding="utf-8" ?>
02 <nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
03
04     <targets>
05         <target name="logfile" xsi:type="File" fileName="log.log"/>
06     </targets>
07
08     <rules>
09         <logger name="*" minLevel="Debug" writeTo="logfile"/>
10     </rules>
11 </nlog>
```

Listing 24 - Beispielhafte NLog-Konfigurationsdatei

Listing 24 zeigt ein Beispiel für eine NLog-Konfigurationsdatei. In diesem Beispiel wird ein einfacher *Logger* (eine Instanz, die Meldungen schreibt) konfiguriert, der in eine Datei namens *„log.log“* schreibt. Die Datei wird als *Target (dt.:“ Ziel“)* bezeichnet. In einer solchen Konfigurationsdatei können beliebig viele verschiedene Targets und Logger definiert werden; der Regelsatz ist sehr umfangreich und erlaubt u.a. auch die formatierte Ausgabe von Log-Meldungen. Es gibt darüber hinaus zahlreiche Möglichkeiten, Targets zu definieren; so ist es u.a. möglich, Log-Meldungen in

[42] http://www.nlog-project.org/
[43] http://www.opensource.org/licenses/bsd-license.php
[44] http://logging.apache.org/log4net/
[45] http://logging.apache.org/log4j/

Dateien, auf die Standard-Ausgabe und in Datenbanken zu schreiben. Eine vollständige Liste der möglichen Targets kann der Website des Herstellers[46] entnommen werden.

Da die Server-Komponente über lediglich eine einzige Konfigurationsdatei verfügen sollte (vgl. 7.5), wurde von der Konfiguration des Logging-Systems über eine zusätzliche Datei abgesehen. Die Konfiguration sollte aber dennoch flexibel genug sein, um Entwicklern, die die Server-Komponente verwenden wollen, die Möglichkeit zu geben, sie ganz nach ihren Vorstellungen zu konfigurieren, ohne die Assembly selbst verändern zu müssen.

Zu diesem Zweck findet die Konfiguration des Logging-Systems nicht in der Assembly *Orchid.Core*, sondern extern statt (in der Referenzimplementierung in der Assembly *Orchid*). Um eine weitere Konfigurationsdatei zu vermeiden, wird NLog aus dem Quellcode der Referenzimplementierung heraus konfiguriert. Das folgende Listing 25 zeigt die Konfiguration aus dem Code heraus, angelehnt an die tatsächliche Verwendung in der Referenzimplementierung (dort werden weitere Code-Teile berührt, die an dieser Stelle nicht relevant sind):

```
01 private static void BuildLoggers()
02 {
03     LogLevel lvl = LogLevel.Debug;
04     LoggingConfiguration config = new LoggingConfiguration();
05
06     FileTarget fileTarget = new FileTarget();
07     fileTarget.FileName = "log.log";
08
09     config.AddTarget(fileTarget);
10
11     LoggingRule rule = new LoggingRule("*", lvl, fileTarget);
12     config.LoggingRules.Add(rule);
13 }
```

Listing 25 - NLog-Konfiguration aus dem Code

Die Konfiguration aus dem Code heraus, die in Listing 25 gezeigt wurde, konfiguriert NLog genau wie die XML-Konfiguration aus Listing 24.

5.5.1.2 Vorteile der Code-Konfiguration

Durch die Konfiguration des Logging-Systems aus dem Code der eigentlichen Server-Anwendung heraus ergibt sich der Vorteil, dass ein Entwickler, der den Code anpassen oder eigenen Code verwenden möchte, die Konfiguration des Logging-Systems vollständig selbst in die Hand nehmen kann. In der Referenzimplementierung wird die Konfiguration des Logging-Systems aus der Konfigurationsdatei für die Server-Komponente ausgelesen; einem Entwickler steht es aber stets frei, eine eigene Konfiguration zu verwenden oder die Konfiguration im Quellcode anzupassen.

Ein zweiter Vorteil, der sich durch den Verzicht auf eine Konfigurationsdatei ergibt, ist die Möglichkeit, die gesamte Konfiguration mit einem Debugger überprüfen zu können. Fehler können so schnell identifiziert und behoben werden.

Weiterhin entspricht es dem Entwurfskriterium der Einfachheit, dass die gesamte Server-Komponente über eine zentrale Datei konfiguriert werden kann – ein Entwickler muss sich lediglich mit dieser einen Datei vertraut machen, um die Komponente nach seinen eigenen Vorstellungen zu konfigurieren.

[46] http://nlog-project.org/wiki/Targets

5.5.1.3 Beispiel für Log-Meldung

An dieser Stelle soll ein Beispiel für Log-Meldungen (in Auszügen) gegeben werden, wie sie für die Server-Komponente des ORCHID Toolkits typisch sind. Für dieses Beispiel wurde die Konfiguration so verändert, dass ein darin angegebener Typ (`ProprietaryDatabaseAdapter`) nicht geladen werden konnte; es handelt sich also um ein Szenario, in dem die Software weiterentwickelt bzw. verändert wurde, der Entwickler jedoch in der Konfiguration einen Fehler gemacht hat. Besonderes Augenmerk liegt beim Logging in einem solchen Fall darauf, dass eine potentielle Fehlerquelle verständlich benannt wird. Die entsprechenden Stellen sind farblich hervorgehoben.

```
(25.08.2011 20.56.03.495) Orchid.Core.Typeloading.NinjectBase..ctor [Debug] ::
Creating Ninject Kernel...

(25.08.2011 20.56.03.596) Orchid.Core.Typeloading.ServerModule.Load [Debug] ::
Injecting dependencies...

(25.08.2011 20.56.03.616) Orchid.Core.Typeloading.ServerModule.GetTypeByName
[Debug] :: Loading 'Orchid.Core.Servers.OrchidHardwareServer' from 'Orchid2.dll'

[...]

(25.08.2011 20.56.03.686) Orchid.Core.Typeloading.ServerModule.GetTypeByName
[Warn] :: Warning: Assembly file
'N:\svn\trunk\Orchid\bin\Debug\SomeCompany.DataAccess.dll' could not be loaded

(25.08.2011 20.56.03.686) Orchid.Core.Typeloading.ServerModule.GetTypeBinding
[Warn] :: Warning: Type 'SomeCompany.DataAccess.ProprietaryDatabaseAdapter' could
not be loaded from 'SomeCompany.DataAccess.dll'

(25.08.2011 20.56.03.716) Orchid.Core.Typeloading.ServerModule.GetTypeBinding
[Info] :: Using Type 'Orchid.Data.MySqlAdapter' instead
```

Listing 26 - Beispielhafte Log-Meldung

Der im obigen Listing gezeigte Auszug aus einer typischen Logdatei der Server-Komponente zeigt deutlich, dass der Fokus der erzeugten Log-Meldungen auf deren Verständlichkeit liegt. Wie einleitend erwähnt, ist die Konfiguration der Server-Komponente, die die obige Log-Meldung erzeugt hat, fehlerhaft, d.h. eine Assemblydatei ist nicht auffindbar; ein Typ, der sich in dieser Datei befinden soll, kann entsprechend nicht geladen werden.

Ein Entwickler, der eine solche Log-Meldung erhält, kann die Ursache des Problems leicht beheben, indem er prüft, ob die Datei vorhanden ist und den angegebenen Typ tatsächlich beinhaltet.

5.5.2 Ninject

Bereits zu Beginn der Entwicklung stand fest, dass ein Dependency Injection Framework zum Einsatz kommen sollte. Besonderes Augenmerk für die Auswahl eines Frameworks lag dabei einerseits auf der möglichst einfachen und intuitiven Verwendung, andererseits auf der Leichtgewichtigkeit; es sollte sich um ein reines Dependency Injection Framework handeln, das keine weiteren Funktionen liefert und keine zusätzlichen Abhängigkeiten zu anderen Frameworks beinhaltet.

Aus dem Umfeld von Java ist das *Spring Framework*[47] sehr bekannt. Eine Portierung für .NET/Mono ist mit *Spring.net*[48] verfügbar. Spring.net erschien für die Verwendung als Dependency Injection Framework in der Server-Komponente jedoch ungeeignet, da es einerseits zahlreiche weitere Funktionen liefert, die nicht verwendet werden würden, andererseits aufwändig in der Konfiguration

[47] http://www.springsource.org/
[48] http://www.springframework.net/

ist. Ähnliche Probleme ergaben sich bei anderen Frameworks: oft ist Dependency Injection nur ein kleiner Teil des Funktionsumfangs dieser Frameworks oder die Konfigurationsaufwände sind sehr hoch.

Die Wahl fiel letztendlich auf *Ninject*[49]. Dieses Dependency Injection Framework wurde nach genau den geforderten Prinzipien entwickelt. Es ist leichtgewichtig, da es keine Abhängigkeiten zu anderen Frameworks beinhaltet und ein reines Dependency Injection Framework, das keine weiteren Funktionen liefert, die es für die vorliegende Arbeit ungeeignet erscheinen lassen würden. Ein weiterer Vorteil, den Ninject mit sich bringt, ist der Verzicht auf große, komplexe Konfigurationsdateien, wie sie z.B. bei Spring.net zum Einsatz kommen. Solche Konfigurationsdateien bergen die Gefahr von Fehlern, die erst zur Laufzeit sichtbar und entsprechend schwer zu beheben sind (das Debugging von Fehlern in Konfigurationsdateien ist immer ein manueller Prozess, da Debugger diese Dateien nicht berücksichtigen).

Ninject ist ein Open-Source-Projekt und für jeden Zweck frei verwendbar, es ergeben sich also keine Lizenzprobleme im Zusammenhang mit der Verwendung im ORCHID Toolkit.

5.5.2.1 Funktionsweise

Da die Funktionsweise von Ninject im weiteren Verlauf dieser Arbeit von zentraler Bedeutung ist (vgl. 6.3.2), soll sie in diesem Abschnitt erklärt werden.

Ninject nutzt zur Dependency Injection einen sog. *„Kernel“ (dt.: Kern)*. Dieser dient als zentrale Anlaufstelle für alle Codeabschnitte, die Objekte über Dependency Injection erzeugen wollen. Abstrahiert dargestellt ist ein Ninject-Kernel also eine sehr allgemein gehaltene Fabrikklasse (vgl. 5.3.1), die eine generische Methode zur Objekterzeugung bereitstellt.

Um einen Ninject-Kernel in die Lage zu versetzen, Objekte zu erzeugen, müssen vor dessen Verwendung die Bindungen zwischen abstrakten Basisklassen und Schnittstellen zu den jeweiligen konkreten Implementierungen bekannt gemacht werden. Dies geschieht über ein sog. *„Modul“*. Ein Modul ist dabei eine beliebige Klasse, die von der von Ninject gelieferten Klasse `NinjectModule` erbt und deren abstrakte Methode `Load()` überschreibt.

Eine Bindung zwischen einer abstrakten Basisklasse oder einer Schnittstelle zu der jeweiligen konkreten Implementierung wird in der `Load()`-Methode des Moduls definiert, wie das folgende Listing 27 zeigt.

```
01 public class CustomModule : NinjectModule
02 {
03     public override void Load()
04     {
05         Bind<AbstractClass>().To(ConcreteType);
06         Bind<InterfaceClass>().To(OtherConcreteType);
07     }
08 }
```

Listing 27 - Beispiel für ein NinjectModule

Die hier gezeigte `Load()`-Methode bindet die konkrete Implementierung `ConcreteType` an die abstrakte Basisklasse `AbstractClass`, sowie die konkrete Implementierung `OtherConcreteType` an die Schnittstelle `InterfaceClass`.

[49] http://www.ninject.org/

Wird ein Ninject-Kernel mit dem in Listing 27 gezeigten Ninject-Modul `CustomModule` konfiguriert, sind ihm die Bindungen für `AbstractClass` und `InterfaceClass` bekannt.

```
01 var modules = new INinjectModule[] { new CustomModule() };
02 StandardKernel kernel = new StandardKernel(modules);
```

Listing 28 - Erzeugung eines Ninject-Kernels

Listing 28 zeigt die Erzeugung eines Ninject-Kernels (`StandardKernel`) mit dem in Listing 27 dargestellten Modul `CustomModule`. Diesem Kernel sind die Bindungen für `AbstractClass` und `InterfaceClass` bekannt, eine Erzeugung von Objekten ist deshalb nun möglich:

```
01 public void TestNinject(StandardKernel kernel)
02 {
03     AbstractClass instance1 = kernel.Get<AbstractClass>();
04
05     InterfaceClass instance2 = kernel.Get<InterfaceClass>();
06 }
```

Listing 29 - Objekterzeugung mit Ninject

Durch die in Listing 27 definierten Bindungen werden im obigen Listing Instanzen der konkreten Implementierungen von `AbstractClass` und `InterfaceClass`, also in diesem Fall `ConcreteClass` und `OtherConcreteClass`, erzeugt.

5.5.2.2 Vorteile

Besonders hervorzuheben ist an dieser Stelle die Einfachheit, mit der Ninject konfiguriert und verwendet werden kann. Andere Dependency Injection Frameworks erfordern zu diesem Zweck einen weitaus größeren Konfigurationsaufwand und sind z.T. auch in der Anwendung wesentlich komplexer.

Ebenso wichtig wie die Einfachheit ist die Robustheit, die Ninject durch den verwendeten Konfigurationsansatz mit sich bringt: der gesamte Konfigurationsprozess kann von einem Debugger überwacht werden, Konfigurationsfehler sind nahezu ausgeschlossen.

Trotz der Leichtgewichtigkeit des Frameworks bietet Ninject alle Vorteile eines Dependency Injection Frameworks, wobei insbesondere die Entkopplung von Klassen innerhalb eines Softwareprojekts an dieser Stelle genannt werden soll.

5.5.3 Nini

Es war von Beginn der Entwicklung an ein Entwurfskriterium, dass die Server-Komponente frei über eine einzelne Datei konfigurierbar sein sollte. Zu diesem Zweck musste ein Framework oder eine Bibliothek gefunden werden, die einerseits flexibel genug für die Anforderungen ist, die die Server-Komponente an die Konfigurierbarkeit stellt, andererseits aber möglichst wenig Overhead in den Code der Komponente einbringt.

Eine eigene Implementierung erschien nicht sinnvoll, da es bereits zahlreiche vorhandene Konfigurations-Bibliotheken auf dem Markt gibt und eine weitere, möglicherweise weitaus weniger robuste Implementierung dem Anti-Pattern *„Reinventing the Wheel*[50]*“* entsprochen hätte.

Da die Server-Komponente plattformübergreifend eingesetzt werden können soll, konnte auch nicht auf die Möglichkeiten zurückgegriffen werden, die Entwicklungsumgebungen wie Microsoft Visual

[50] *„Das Rad neu erfinden“*, ein Anti-Pattern, das eine eigene (oft ineffiziente) Implementierung einer Funktion beschreibt, die bereits verfügbar ist.

Studio 2010 einem Entwickler bieten. In dieser Entwicklungsumgebung, die auf die Entwicklung mit dem .NET Framework ausgerichtet ist, können Anwendungseinstellungen leicht über Funktionen des .NET Frameworks vorgenommen und verwaltet werden – es besteht aber stets die Gefahr einer Inkompatibilität zu Mono, weshalb diese Vorgehensweise ebenfalls ungeeignet erschien.

In der Server-Komponente wird deshalb das unter der MIT-Lizenz[51] veröffentlichte Konfigurations-Framework *Nini*[52] verwendet. Nini ist ein Open-Source-Projekt und kann aufgrund der verwendeten Lizenz problemlos in die Server-Komponente eingebunden werden.

Nini ist in der Lage, zahlreiche verschiedene Quellen für die Konfiguration einer Anwendung zu lesen und zu schreiben, darunter u.a. INI-Dateien, XML-Dateien und Kommandozeilenparameter. Dennoch handelt es sich um eine sehr kleine Bibliothek, die keine tiefgreifenden Auswirkungen auf den restlichen Code der Server-Komponente hat. Darüber hinaus ist die Verwendung von Nini sehr einfach und flexibel, wie im Folgenden exemplarisch gezeigt werden soll.

5.5.3.1 Konfigurationsdateien mit Nini verwenden

Die Verwendung von Konfigurationsdateien mit Nini ist denkbar einfach gehalten und erfordert seitens eines Entwicklers lediglich einen geringen Aufwand.

Nini stellt die Schnittstelle `IConfigSource` und verschiedene davon abgeleitete Klassen zur Verfügung, die jeweils eine bestimmte Art von Konfigurationsquelle verarbeiten können. Um eine Konfigurationsdatei nutzen zu können, muss lediglich eine neue Instanz einer von `IConfigSource` abgeleiteten Klasse erzeugt und dieser der Pfad zu der zu verwendenden Konfigurationsquelle genannt werden.

```
01 // INI file
02 IConfigSource iniSource = new IniConfigSource("config.ini");
03
04 // XML file
05 IConfigSource xmlSource = new XmlConfigSource("config.xml");
```

Listing 30 - Laden einer Konfigurationsdatei mit Nini

Listing 30 zeigt die Initialisierung zweier Konfigurationsquellen. Die Quelle `iniSource` ist vom Typ `IniConfigSource` und repräsentiert eine Konfigurationsdatei im INI-Format (Details zu INI-Dateien können Abschnitt 7.5 entnommen werden), `xmlSource` hingegen ist vom Typ `XmlConfigSource` und repräsentiert entsprechend eine Datei im XML-Format.

Liegt eine Instanz einer von `IConfigSource` abgeleiteten Klasse vor, können Konfigurationselemente aus der übergebenen Konfigurationsdatei ausgelesen werden. Für das folgende Beispiel wird angenommen, dass eine INI-Datei mit dem im folgenden Listing gezeigten Aufbau als Konfigurationsquelle übergeben wird.

```
01 ;Example configuration (INI file)
02 [Example]
03 Entry1 = "Demonstration"
04 Entry2 = 1234
05
06 [Others]
07 Entry1 = "Another Demonstration"
```

Listing 31 - Beispiel für eine INI-Datei

[51] http://www.opensource.org/licenses/mit-license.php
[52] http://nini.sourceforge.net/

Die Einträge der Konfigurationsdatei können über eine Instanz einer von *IConfigSource* abgeleiteten Klasse ausgelesen werden, wie in Listing 32 gezeigt wird. Dabei müssen der abfragenden Methode die Namen der Kategorie und des gesuchten Elements übergeben werden. Aus diesem Grund ist es auch möglich, dass zwei Elemente mit dem gleichen Namen in verschiedenen Kategorien existieren können (im Beispiel ist dies für *„Entry1“* der Fall).

```
01 private void ConfigExample(string file)
02 {
03     IConfigSource src = new IniConfigSource(file);
04
05     // Will contain "Demonstration"
06     string entry1_example = src.Configs["Example"].Get("Entry1");
07
08     // Will contain "1234" (int)
09     int entry2_example = src.Configs["Example"].GetInt("Entry2");
10
11     // Will contain "Another Demonstration"
12     string entry1_others  = src.Configs["Others"].Get("Entry1");
13 }
```

Listing 32 - Verwendung von Nini

Das Beispiel im obigen Listing zeigt, wie *IConfigSource* die Struktur der verwendeten Konfigurationsdatei abbildet: die Instanz *src* stellt eine Eigenschaft *Configs* bereit (es handelt sich um einen sog. *„Indexer“*, also eine *Property* (vgl. 5.4.1.1), die einen Parameter erwartet), der als Parameter der Name der Sektion übergeben werden muss, aus der ein Eintrag gelesen werden soll. *Configs* ist eine Sammlung von Konfigurations-Sektionen. Eine solche Sektion stellt ihrerseits die Methode *Get* () und weitere Varianten, z.B. *GetInt()*, zur Verfügung, die als Parameter den Namen des Konfigurationselements erwarten, das gelesen werden soll.

5.5.3.2 Vorteile

Der Hauptvorteil, den Nini einem Entwickler bietet, ist meiner Meinung nach die Simplizität, mit der das Framework verwendet werden kann. Anstatt ein eigenes Konfigurationssystem implementieren zu müssen, kann ein Entwickler innerhalb weniger Codezeilen eine Konfigurationsdatei einlesen und die darin enthaltenen Elemente zur Konfiguration einer Software nutzen.

Durch die verschiedenen Methoden, die eine Konfigurations-Sektion anbietet, um Daten aus einer Konfigurationsdatei zu lesen, werden grobe Konfigurationsfehler vermieden, da es zu einer Fehlermeldung kommt, wenn ein auszulesender Wert nicht in den Typ umgewandelt werden kann, der von der entsprechenden Methode erwartet wird (im Beispiel in Listing 32 wäre dies z.B. der Fall, wenn für *„Entry2“* ein Wert angegeben worden wäre, der nicht in einen Integer-Wert umgewandelt werden kann).

5.5.3.3 Nachteile

Die Verwendung von Nini hat keinen negativen Einfluss auf die Entwicklung oder das Verhalten der Software, für deren Konfiguration es eingesetzt wird.

Im Gegensatz zu den einleitend erwähnten Konfigurationsmöglichkeiten, die Microsoft Visual Studio einem Entwickler bietet, verfügt Nini aber nicht über die Möglichkeit, vorhandene Einstellungs-Elemente als typsichere Eigenschaften einer global verfügbaren, zentralen Klasse bereitzustellen, wodurch Typprüfungen bereits durch den Compiler, nicht erst zur Laufzeit, durchgeführt werden könnten.

5.5.4 Microsoft FxCop

Microsoft FxCop[53] (*„Microsoft Framework Cop“*) ist ein kostenloses Tool, das in der Lage ist, CLI - Assemblies zu analysieren und die Einhaltung bestimmter Regeln, der sog. *„Design Guidelines for Class Library Developers“*[54] (kurz: *Design Guidelines*) durch den Entwickler der Assembly zu überprüfen.

Die Design Guidelines umfassen 196 Regeln aus den Bereichen *Design, Globalization, Interoperability, Mobility, Naming, Performance, Portability, Security* und *Usage*, wobei FxCop es erlaubt, einzelne Regeln bei der Analyse einer Assembly nicht anzuwenden, wenn dies gewünscht ist.

Für die Analyse der Server-Komponente wurden 191 der 196 Regeln herangezogen. Diejenigen Regeln, die nicht betrachtet wurden, werden im Folgenden kurz erläutert und der Grund für ihre Entfernung genannt.

Nr.	Regel	Grund für Entfernung
1	*Avoid namespaces with few types*	Eine saubere semantische Trennung wurde der Einhaltung der Regel vorgezogen.
2	*Do not catch general exception types*	Obwohl während der Entwicklung besonderes Augenmerk auf die Fehlerbehandlung gesetzt wurde, fängt der Code sicherheitshalber auch allgemeine Exceptions ab, um Programmabstürze zu vermeiden.
3	*Identifiers should have correct suffix*	Die Einhaltung dieser Regel würde zu Typnamen führen, die überflüssige Namensbestandteile tragen. Das Auslassen der Regel führt nicht zu Kompatibilitätsproblemen.
4	*Identifiers should not contain type names*	Würde die Regel eingehalten, müssten zahlreiche Methoden, deren Namen die Methode selbsterklärend machen, in einer Art umbenannt werden, die einen Entwickler eher verwirren würde. Das Auslassen dieser Regel führt nicht zu Kompatibilitätsproblemen.
5	*Do not call overridable methods in constructors*	Die Klassen der Server-Komponente bestehen zum größten Teil aus überschreibbaren oder virtuellen Methoden. Die Einhaltung dieser Regel würde zu zahlreichen Code-Dopplungen führen, die die Wartbarkeit des Codes stark einschränken würden.

Tabelle 66 - Nicht verwendete FxCop-Regeln

5.5.5 Microsoft StyleCop

Das von Microsoft als Open-Source-Projekt veröffentlichte Tool *StyleCop*[55] dient der statischen Codeanalyse von Code für das .NET Framework, mit dem auch Code für die Plattform Mono analysiert werden kann.

Ähnlich wie das im vorherigen Abschnitt beschriebene FxCop prüft auch StyleCop die Einhaltung zahlreicher Regeln. StyleCop umfasst 139 Regeln aus den Bereichen *Documentation, Layout,*

[53] http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx
[54] http://msdn.microsoft.com/de-de/library/czefa0ke(v=vs.71).aspx
[55] http://stylecop.codeplex.com/

Maintainability, Naming, Ordering, Readability und *Spacing*. Schon aus den Namen der Kategorien geht deutlich hervor, dass die Einhaltung der von StyleCop überprüften Regeln zu sauberem Code führt, der von anderen leicht verstanden und eingesetzt werden kann.

Insbesondere die Kategorie *„Documentation“* sei an dieser Stelle erwähnt, da die in ihr enthaltenen Regeln dazu führen, dass der entsprechende Code lückenlos dokumentiert ist. Diese Kategorie enthält 45 Regeln zur Dokumentation von Code mittels XML-Kommentaren. Angefangen bei groben Verstößen, z.B. wenn eine Methode gänzlich undokumentiert ist (Regel *SA1600*, *„Elements Must Be Documented“*), bis hin zu feingranularen Regelbrüchen (Regel *SA1629*, *„Documentation Text Must End With A Period“*) wird die umfangreiche Dokumentation von Code durch StyleCop forciert. Werden alle Dokumentationsregeln ordnungsgemäß eingehalten, kann über entsprechende Werkzeuge eine umfangreiche Programm-Dokumentation erzeugt werden, die jeden Aspekt des Programms dokumentiert.

Wie auch FxCop erlaubt StyleCop, einzelne Regeln auszublenden. Für die Analyse wurden 137 der 139 Regeln angewendet und eingehalten. Die zwei nicht verwendeten Regeln und der Grund für ihre Auslassung werden in Tabelle 67 gezeigt.

Nr.	Regel	Grund für die Auslassung
SA1309	Field names must not contain underscores.	Die Kennzeichnung privater Klassenvariablen mit einem einleitenden Unterstrich führt zu einer besseren optischen Trennung zwischen Klassenvariablen und lokalen Variablen, z.B. in einer Methode. Die Auslassung dieser Regel führt nicht zu Kompatibilitätsproblemen.
SA1200	Using directives must be placed within the namespace.	In Programmiersprachen, die über Pakete verfügen, ist es üblich, using-Direktiven (in Java: import) an den Anfang einer Datei zu stellen. Diese Vorgehensweise wird auch im Code der Server-Komponente angewendet.

Tabelle 67 - Nicht verwendete StyleCop-Regeln

5.6 Klassen der Server-Komponente

Im Folgenden werden die wesentlichen Klassen der Server-Komponente auf Basis der in 4.1.1 definierten Anforderungen entwickelt. Es handelt sich dabei nicht um eine vollständige Auflistung aller in der Server-Komponente enthaltenen Klassen, sondern um eine Hervorhebung der wichtigsten Bestandteile. Eine vollständige Liste kann der DVD, die dieser Arbeit beiliegt, entnommen werden.

5.6.1 Server und Daemons

Server und *Daemons* sind die zentralen Klassen zur Annahme und Behandlung von Anfragen eines Clients oder einer Drittsoftware an einen ORCHID Server. Server bilden dabei die Schnittstelle zum Netzwerk bzw. Internet, während Daemons für die eigentliche Anfragebehandlung zuständig sind.

Es existieren je zwei Arten von Servern und Daemons: *Hardware-Server* und *Hardware-Daemons* sind für die Annahme und Behandlung von Client-Anfragen zuständig, während *Software-Server* und *Software-Daemons* Anfragen von Drittsoftware entgegennehmen und verarbeiten.

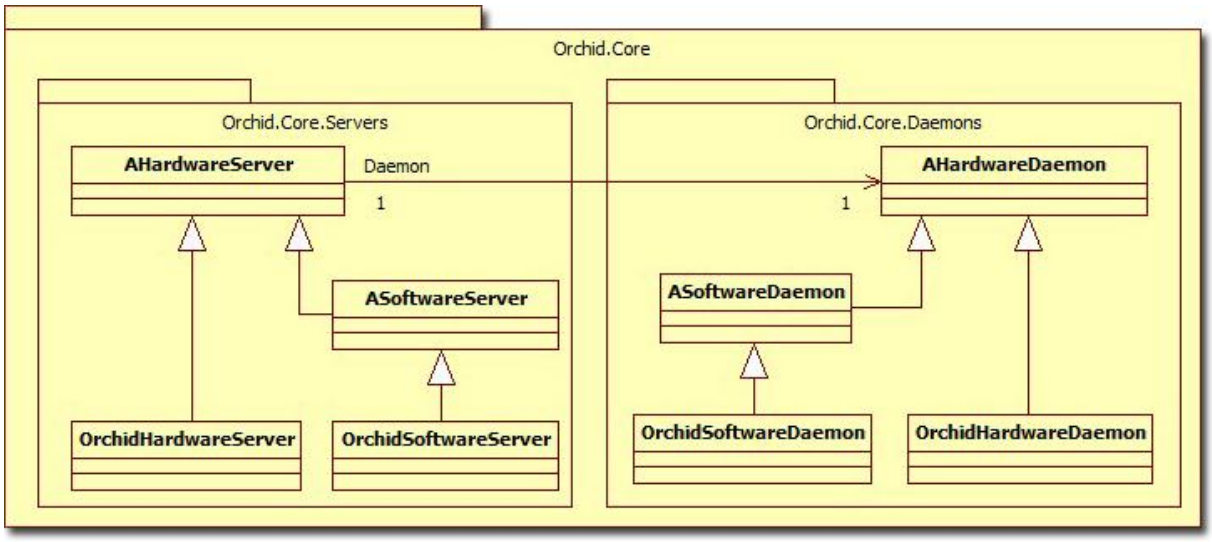

Abbildung 16 - Vererbungshierarchie von Servern und Daemons

Wie die obige Abbildung darstellt, werden Server stets von der abstrakten Basisklasse `AHardwareServer` abgeleitet und enthalten immer einen ihnen zugeordneten Daemon; jeder Daemon leitet von der abstrakten Basisklasse `AHardwareDaemon` ab und ist immer genau einem Server zugeordnet.

Obwohl es sich bei den Klassen `AHardwareServer` und `ASoftwareServer`, bzw. `AHardwareDaemon` und `ASoftwareDaemon` jeweils effektiv um die gleiche Klasse handelt (`ASoftwareServer` und `ASoftwareDaemon` enthalten keine Methoden oder Eigenschaften, die ihre jeweilige Basisklasse nicht enthält), wurde die Trennung in je eine eigene Klasse vorgenommen, um spätere Veränderungen und Erweiterungen feingranular zu ermöglichen.

Als zentrale Bestandteile der Server-Komponente nutzen *Daemon*-Klassen zahlreiche weitere Klassen, um die ihnen zugewiesenen Aufgaben zu erledigen. Jede Instanz eines Daemons verfügt über eine Liste von Client-Berechtigungen (`Credentials`), eine verifizierende Instanz (`Verifier`), die zur Validierung eingehender Anfragen genutzt wird, Fabriken zur Erzeugung von Datenhaltungsobjekten (`DataFactory` und `HardwareFactory`), eine Responder-Instanz (`Responder`), sowie eine Parser-Instanz (`Parser`) und eine Instanz eines Datenbankadapters (`Database`). Die nachfolgende Abbildung 17 zeigt die Beziehungen einer Daemon-Klasse anhand der abstrakten Basisklasse

AHardwareDaemon (auf die Darstellung von Namespaces wurde aufgrund der besseren Übersicht verzichtet).

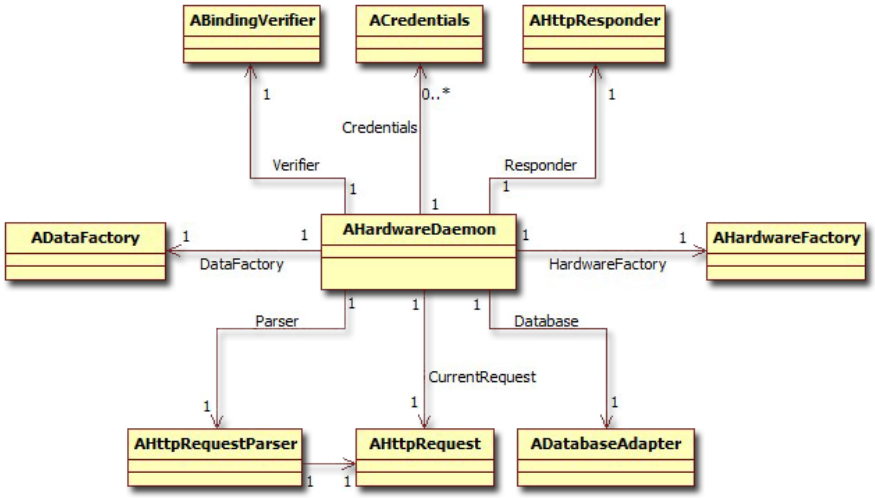

Abbildung 17 - Beziehungen der Daemon-Klasse

5.6.2 Parser und Requests

Clients und Drittsoftware übermitteln Daten als Strom von Bytes. Um Anfragen komfortabel und fehlerfrei zu verarbeiten, werden eingehende Datenströme durch eine *Parser*-Instanz in eine *Request*-Instanz überführt.

Die abstrakte Klasse *AHttpRequestParser* dient als Grundlage für alle Parser-Instanzen und erzeugt Objekte der Klasse *AHttpRequest*, die wiederum die abstrakte Basisklasse für alle Request-Instanzen ist.

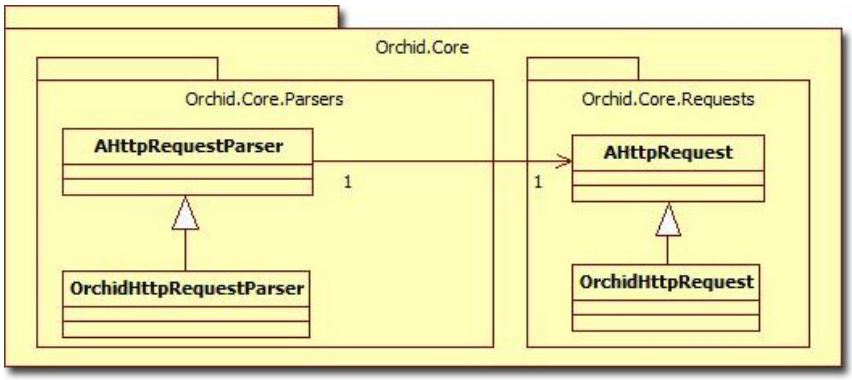

Abbildung 18 - Vererbungshierarchie von Parsern und Requests

Die obige Abbildung zeigt die Vererbungshierarchie in einem UML-Klassendiagramm.

5.6.3 Datenhaltungsklassen

Die Server-Komponente beinhaltet zahlreiche Datenhaltungsklassen und zusätzlich entsprechende Fabriken, um Instanzen dieser Klassen zu erzeugen. Als Basis kommt eine Schnittstelle namens *IGuid* zum Einsatz, die von allen Datenhaltungsklassen implementiert wird (mit Ausnahme von *ASensorData*). Auf diese Weise verfügen alle Objekte implementierender Klassen über eine GUID, mit der sie eindeutig zu identifizieren sind.

Die folgende Abbildung 19 zeigt Beziehungen und Hierarchien der Datenhaltungsklassen untereinander und in Verbindung mit den entsprechenden Fabriken.

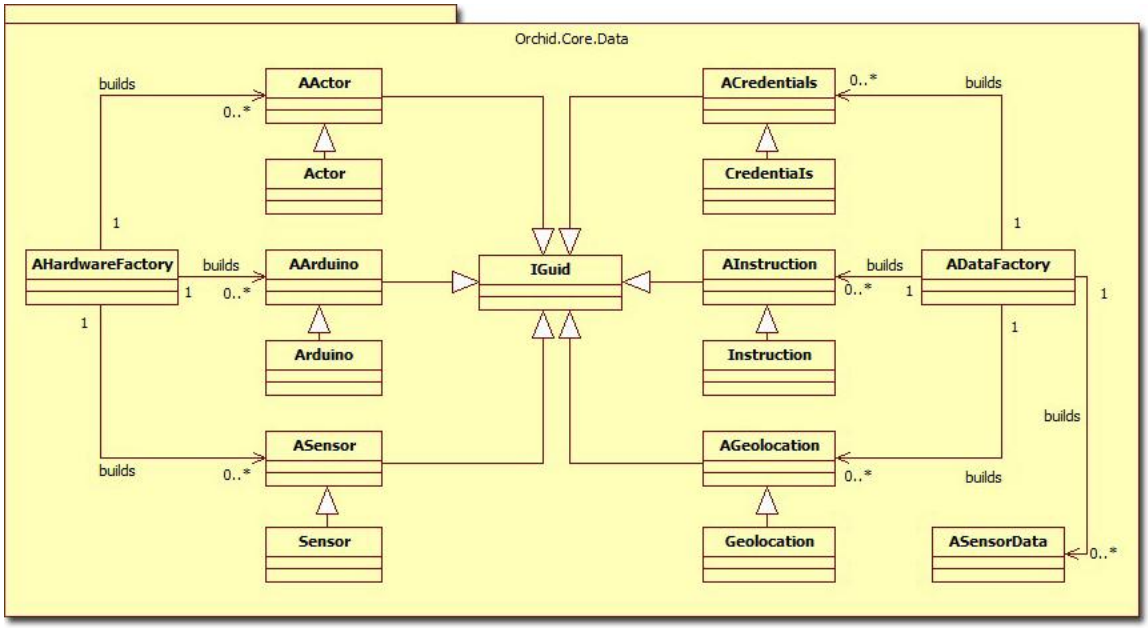

Abbildung 19 - Beziehungen und Hierarchien der Datenhaltungsklassen

Da es durch die Vorgaben des Entwurfsmusters *„Abtract Factory Pattern“* (vgl. 5.3.1) notwendig ist, eine direkte Erzeugung von Objekten zu verhindern, die von Fabrikklassen erzeugt werden sollen, verfügen die dargestellten Datenhaltungsklassen nicht über öffentliche (`public`) Konstruktoren. Nur Klassen im gleichen Namespace (`Orchid.Core.Data`) oder erbende Klassen können die Konstruktoren der abgebildeten Klassen aufrufen, da diese mit dem Zugriffsmodifizierer `protected internal` ausgestattet sind. Deshalb befinden sich auch die Fabrikklassen `ADataFactory` und `AHardwareFactory` im Namespace `Orchid.Core.Data`.

5.6.4 Datenbankadapter

Zur Anbindung an eine Datenbank oder einen anderen Mechanismus zur Datenspeicherung nutzt die Server-Komponente die abstrakte Klasse `ADatabaseAdapter`. Diese Klasse stellt zahlreiche Methoden zur Verfügung, um verschiedene Operationen auf der zugrunde liegenden Datenbank durchzuführen. Da diese Methoden häufig Objekte von anderen Klassen der Server-Komponente zurückgeben, existieren zahlreiche Beziehungen zwischen `ADatabaseAdapter` und anderen Klassen, die in der nachstehenden Abbildung 20 dargestellt werden.

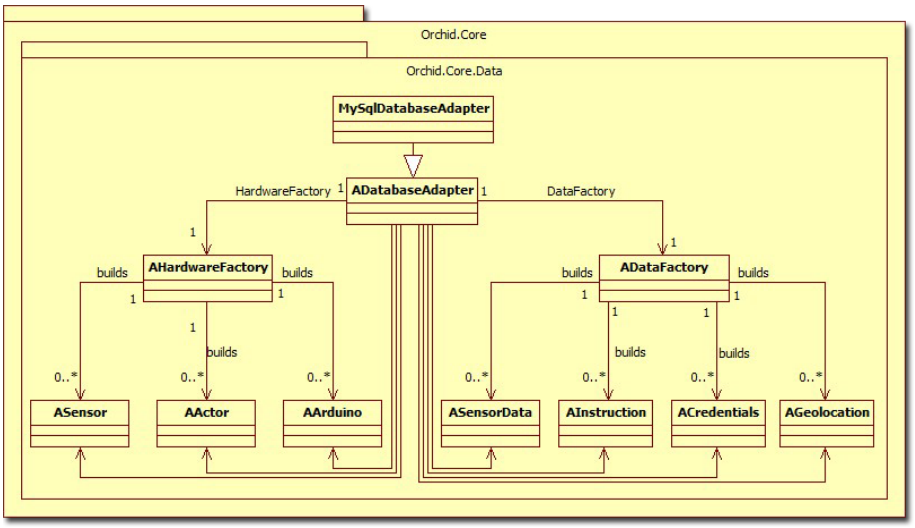

Abbildung 20 - Beziehungen der Datenbankadapter-Klasse

5.6.5 Dynamisches Typeloading

Das dynamische Typeloading wird in 6.3.2 im Detail erläutert und an dieser Stelle nur aus Gründen der Vollständigkeit erwähnt.

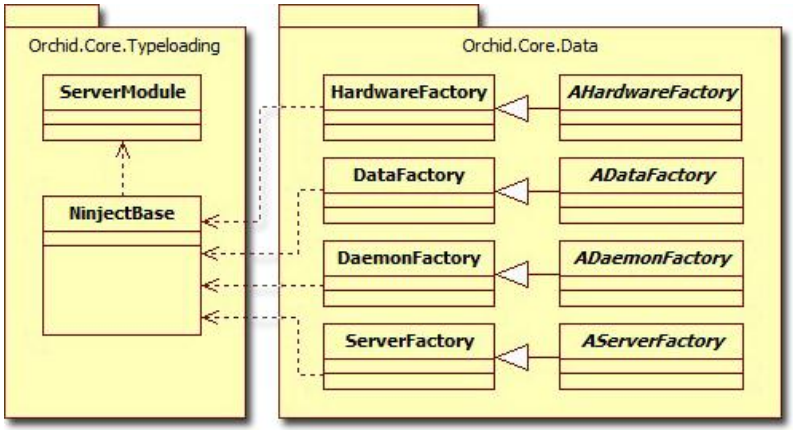

Abbildung 21 - Abhängigkeiten des Typeloading-Systems

Die obige Abbildung zeigt die Abhängigkeiten derjenigen Klassen, die maßgeblich am Prozess des dynamischen Typeloadings in der Server-Komponente beteiligt sind. Um Entwicklern die Möglichkeit zu geben, die Server-Komponente ohne eine Bindung an das verwendete Dependency Injection Framework (Ninject, vgl. 5.5.2) erweitern und verändern zu können, gibt es keine Abhängigkeiten abstrakter Basisklassen an die von Ninject bereitgestellten Klassen. Lediglich die konkreten Implementierungen der Fabrikklassen (`HardwareFactory`, `DataFactory`, `DaemonFactory` und `ServerFactory`) nutzen Ninject über die Klasse `NinjectBase`, um Objekte mittels Dependency Injection zu erzeugen.

5.7 Klassen der Client-Bibliothek

Die Client-Bibliothek besteht aus lediglich zwei Klassen, die in Sketches für einen Arduino-Microcontroller verwendet werden können: `Orchid` und `OrchidInstruction`.

Die Klasse `Orchid` bildet den zentralen Kern der Client-Bibliothek und stellt einem Entwickler Funktionen zur Verfügung, um mit einem ORCHID Server zu kommunizieren. `OrchidInstruction` kapselt Instruktionen, die von einem Server für den ausführenden ORCHID Client abgerufen wurden, und stellt zusätzlich Hilfsmethoden bereit.

Die folgende Abbildung 22 zeigt die Klassen der Client-Bibliothek in einem UML-Klassendiagramm.

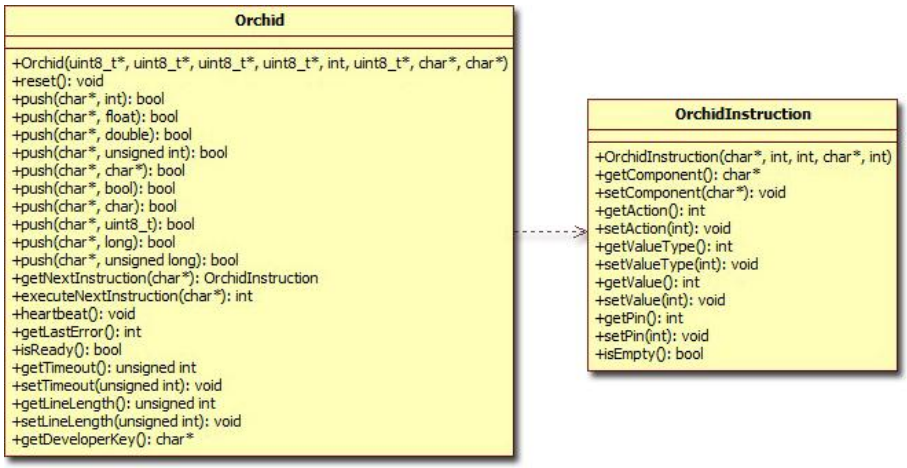

Abbildung 22 - Klassen der Client-Bibliothek

Die in der obigen Abbildung gezeigte Abhängigkeit führt dazu, dass bei Einbindung der Klasse `Orchid` auch `OrchidInstruction` in einen Sketch eingebunden werden muss.

5.8 Protokoll

Die Kommunikation zwischen Server-Komponente, Client-Bibliothek und Drittsoftware geschieht über HTTP-Nachrichten, die zwischen den einzelnen Kommunikationsteilnehmern versendet werden. Da mehrere verschiedene Technologien zum Einsatz kommen[56], um Anfragen zu versenden, zu verarbeiten und zu beantworten, ist es besonders wichtig, dass sich alle beteiligten Stellen auf ein gemeinsames Protokoll einigen, von dessen Einhaltung jeder Kommunikationsteilnehmer ausgehen kann.

Das vom ORCHID Toolkit verwendete Protokoll basiert auf HTTP in der Version 1.1 und muss strikt eingehalten werden, damit das Toolkit wie in dieser Arbeit beschrieben funktionieren kann und Fehler vermieden werden können.

5.8.1 Allgemeines

HTTP nutzt, wie in 2.2 bereits erwähnt, zur Datenübertragung sowohl die Header-Felder, als auch den Message Body, also eine Zeichenkette, die auf die Header-Felder folgt.

Das vom ORCHID Toolkit verwendete Protokoll setzt für Anfragen an einen ORCHID Server verstärkt auf den Einsatz der Header-Felder, in Antworten des Servers wird aber zusätzlich der Message Body verwendet, da hier auch größere Datenmengen untergebracht werden können.

[56] Mindestens C# (Mono) und C++, optional weitere.

Header-Felder werden untereinander durch die Zeichen `\r\n` (*Carriage Return* gefolgt von *Newline*) voneinander getrennt. Die Trennung von Message Body und Header-Feldern erfolgt ebenfalls durch `\r\n`.

Ein ORCHID Server ist durch das Protokoll dazu verpflichtet, auf eine Anfrage eine Antwort zu senden, falls dies technisch möglich ist.

5.8.2 Mögliche Antworten eines Servers

Nachdem eine Instanz eines ORCHID Servers eine eingegangene HTTP-Anfrage verarbeitet hat, bzw. wenn die Verarbeitung einer Anfrage aufgrund eines Fehlers abgebrochen wurde, sendet sie eine Antwort an den Kommunikationspartner, der die Anfrage gestellt hat. Diese Antwort enthält einen Statuscode und eine Erfolgs- oder Fehlermeldung, falls keine Daten vom Server abgerufen werden sollten oder konnten, oder die angeforderten Daten selbst.

Eine gültige Anfrage eines Kommunikationspartners wird durch den Server stets mit HTTP-Code *200 („OK“)* beantwortet. Wurden keine Daten abgefragt, besteht der Message Body aus einer kurzen Bestätigungsnachricht, ansonsten werden die angeforderten Daten übertragen.

Wurde eine ungültige Anfrage an den Server gesendet (also eine Anfrage, die der Server nicht verarbeiten kann oder die fehlerhafte Daten enthält), antwortet dieser mit HTTP-Code *400 („Bad Request“)* und einer kurzen Fehlermeldung im Message Body.

Kann sich ein Kommunikationspartner nicht ordnungsgemäß beim Server authentifizieren (z.B. bei Übertragung eines nicht akzeptierten Developer Keys, vgl. 6.3.3.2), wird als Antwort HTTP-Code *401 („Unauthorized“)* gesendet und im Message Body eine kurze Fehlermeldung übertragen.

Tritt hingegen ein interner Fehler auf, der durch den Server selbst verursacht wurde, wird die Anfrage mit HTTP-Code *500 („Internal Server Error“)* und einer kurzen Fehlermeldung im Message Body beantwortet.

Antworten eines ORCHID Servers sind immer wie im folgenden Listing gezeigt aufgebaut:

```
01 HTTP/1.1 <HTTP-CODE>
02 Connection: close
03
04 <BODY>
```

Listing 33 - Allgemeine Antwort eines ORCHID Servers

Der Platzhalter *<HTTP-CODE>* wird dabei durch einen der oben beschriebenen Fehler- oder Erfolgscodes ersetzt, *<BODY>* durch eine Meldung oder die abgerufenen Daten, falls solche vorliegen.

In den folgenden Abschnitten werden die verschiedenen Arten von Anfragen an einen ORCHID Server im Detail beschrieben. Die Antworten des Servers werden nur erläutert, wenn Daten übertragen werden, da sie ansonsten dem obigen Listing entsprechen.

5.8.3 Sensordaten hochladen

Ein Client kann Sensordaten auf einen ORCHID Server laden, indem er eine *PUT*-Anfrage an den Server sendet. Die benötigten Header-Felder dieser Anfrage, sowie deren Aufbau, werden in der folgenden Tabelle gezeigt.

Header-Feld	Übertragener Wert
Board-ID	GUID des Clients, der die Anfrage an den Server stellt.
Sensor-ID	GUID des Sensors, der die Daten ermittelt hat.
User-Agent	Zeichenkette, die den Kommunikationspartner als ORCHID Client oder mit dem Toolkit kompatible Drittsoftware identifiziert.
Host	Die IP-Adresse des Servers und der Port, auf dem sich der Client verbinden möchte.
Authorization	Der Developer Key, der verwendet werden soll.
Data	Der Sensorwert, der zu übertragen ist, und dessen Datentyp, jeweils als Zeichenkette.
Allgemeine Form	

`PUT HTTP/1.1`
`Board-ID: <GUID>`
`Sensor-ID: <GUID>`
`User-Agent: <STRING>`
`Host: <IP>:<PORT>`
`Authorization: <DEVELOPER KEY>`
`Data: <TYPE>/<VALUE>`
`Connection: close`	

Beispiel	

`PUT HTTP/1.1`
`Board-ID: 9f476a50-d4e3-11e0-801d-001d92396635`
`Sensor-ID: f6f9a4c0-dae6-4fcc-a1d2-5db57feac465`
`User-Agent: Orchid Arduino Toolkit`
`Host: 192.168.1.123:29100`
`Authorization: cd0887447e04c15b5128629f04b53c4c4858bb1e8ff2e8b6bfcb32dc99fb4e7198d72ed2`
`Data: INT/123`
`Connection: close`	

Tabelle 68 - Protokoll: Sensordaten hochladen

Der in der obigen Tabelle gezeigte PUT-Request lädt den Sensorwert *123* (Integer), gemessen durch den Sensor mit der GUID *f6f9a4c0-dae6-4fcc-a1d2-5db57feac465*, der an den Client mit der GUID *9f476a50-d4e3-11e0-801d-001d92396635* angeschlossen ist, auf einen ORCHID Server, der unter der IP-Adresse *192.168.1.123* und Port *29100* erreichbar ist.

Das ORCHID Toolkit unterstützt eine Vielzahl von Datentypen, die innerhalb eines PUT-Requests übermittelt werden können (weitere Datentypen können nach Bedarf zusätzlich implementiert werden). Tabelle 69 bildet diese Datentypen in alphabetischer Reihenfolge ab.

Datentyp-Code	Mögliche Werte
BOOL	Wahrheitswerte (`boolean`)
BYTE	Vorzeichenlose Ganzzahlen (`unsigned 8-bit Integer`)
CHR	Einzelnes Zeichen (`16-bit`)

Datentyp-Code	Mögliche Werte
DOUBLE	Fließkommazahlen (`64-bit`)
FLOAT	Fließkommazahlen (`32-bit`)
INT	Ganzzahlen (`32-bit Integer`)
LONG	Ganzzahlen (`64-bit Integer`)
STR	Zeichenketten
UINT	Vorzeichenlose Ganzzahlen (`unsigned 32-bit Integer`)
ULONG	Vorzeichenlose Ganzzahlen (`unsigned 64-bit Integer`)

Tabelle 69 - Codes für Datentypen

5.8.4 Heartbeat senden

Möchte ein Client einem Server seine Bereitschaft signalisieren, kann er dies durch die Übertragung einer *TRACE*-Anfrage tun. Ein solcher Request wird auch als *„Heartbeat-Request“* oder kurz *„Heartbeat“ (dt.: „Herzschlag“* oder *„Puls“*) bezeichnet. Die benötigten Header-Felder und der Aufbau eines Heartbeats können der folgenden Tabelle entnommen werden.

Header-Feld	Übertragener Wert
Board-ID	GUID des Clients, der die Anfrage an den Server stellt.
User-Agent	Zeichenkette, die den Kommunikationspartner als ORCHID Client oder mit dem Toolkit kompatible Drittsoftware identifiziert.
Host	Die IP-Adresse des Servers und der Port, auf dem sich der Client verbinden möchte.
Authorization	Der Developer Key, der verwendet werden soll.

Allgemeine Form

```
TRACE HTTP/1.1
Board-ID:       <GUID>
User-Agent:     <STRING>
Host:           <IP>:<PORT>
Authorization:  <DEVELOPER KEY>
Connection:     close
```

Beispiel

```
TRACE HTTP/1.1
Board-ID: 9f476a50-d4e3-11e0-801d-001d92396635
User-Agent: Orchid Arduino Toolkit
Host: 192.168.1.123:29100
Authorization: cd0887447e04c15b5128629f04b53c4c4858bb1e8ff2e8b6bfcb32dc99fb4e7198d72ed2
Connection: close
```

Tabelle 70 - Protokoll: Heartbeat senden

Mit diesem Request meldet der durch die GUID *9f476a50-d4e3-11e0-801d-001d92396635* identifizierte Client einem ORCHID Server, der unter der IP-Adresse *192.168.1.123* und Port *29100* erreichbar ist, seine Bereitschaft.

5.8.5 Instruktion abrufen

Ein Client kann Instruktionen für einen an ihn angeschlossenen Aktor abrufen, indem er einen GET-Request an einen ORCHID Server sendet. Die für diese Art von Anfrage benötigten Header-Felder und ihr Aufbau werden in Tabelle 71 dargestellt.

Header-Feld	Übertragener Wert
Board-ID	GUID des Clients, der die Anfrage an den Server stellt.
User-Agent	Zeichenkette, die den Kommunikationspartner als ORCHID Client oder mit dem Toolkit kompatible Drittsoftware identifiziert.
Host	Die IP-Adresse des Servers und der Port, auf dem sich der Client verbinden möchte.
Authorization	Der Developer Key, der verwendet werden soll.

Allgemeine Form

```
GET /<GUID>/<GUID> HTTP/1.1
Board-ID:       <GUID>
User-Agent:     <STRING>
Host:           <IP>:<PORT>
Authorization:  <DEVELOPER KEY>
Connection:     close
```

Beispiel

```
GET /9f476a50-d4e3-11e0-801d-001d92396635/b0ce2688-d56a-11e0-85d1-001d92396635 HTTP/1.1
Board-ID: 9f476a50-d4e3-11e0-801d-001d92396635
User-Agent: Orchid Arduino Toolkit
Host: 192.168.1.123:29100
Authorization: cd0887447e04c15b5128629f04b53c4c4858bb1e8ff2e8b6bfcb32dc99fb4e7198d72ed2
Connection: close
```

Tabelle 71 - Protokoll: Instruktion abrufen

Aus der obigen Tabelle wird ersichtlich, dass GET-Anfragen eines Clients an einen ORCHID Server von der Form bisher gezeigter Anfragen abweichen. Die Pfadangabe nach der Nennung der Methode (in der ersten Zeile des oben gezeigten Requests) bestimmt, für welchen Client (erste GUID) und welchen Aktor (zweite GUID) eine Instruktion abgerufen werden soll. Im oben gezeigten Beispiel soll eine Instruktion für den Client mit der GUID `9f476a50-d4e3-11e0-801d-001d92396635` und den Aktor mit der GUID `b0ce2688-d56a-11e0-85d1-001d92396635` abgerufen werden.

Diese Form der Anfrage ist z.B. für spätere Erweiterungen nützlich, da auch komplexere Hierarchien als die oben gezeigte problemlos abgebildet werden können.

Neben den einleitend erwähnten, allgemeinen Möglichkeiten für den HTTP-Code der Antwort des Servers, kann das Versenden eines GET-Requests zu zwei weiteren Arten von Antworten führen.

Liegt auf dem Server keine bisher unbearbeitete Anfrage für den anfragenden Client vor, wird als Antwort HTTP-Code 404 (*„Not Found“*) und eine kurze Meldung gesendet. Konnte eine Instruktion ermittelt werden, wird diese über den Message-Body der Antwort mit HTTP-Code 200 (*„OK“*) übermittelt, wie das folgende Listing zeigt.

```
01 HTTP/1.1 200 OK
02 Connection: close
03 Content-Length: 42
04
05 action=1666&valuetype=1114&value=LOW&pin=8
```

Listing 34 - Übermittlung einer Instruktion (Server-Antwort)

Die Instruktion wird als Reihung von Schlüssel-Wert-Paaren (jeweils durch *„&“* getrennt) übertragen.

Auf Seite des Clients muss diese Art der Übertragung bekannt sein, damit die Instruktion verwertet werden kann. Reihenfolge und Schreibweise sind in der aktuellen Version der Client-Bibliothek unbedingt einzuhalten (vgl. 6.6.2).

Die durchzuführende Aktion und der Typ des in der Instruktion enthaltenen Werts (Felder `action` und `valuetype`) sind als Ganzzahlen kodiert. Die Client-Bibliothek kennt die in den folgenden Tabellen beschriebenen Aktionen und Datentypen.

Code	Aktion
1666	Digital Write
2666	Digital Read
3666	Analog Write
4666	Analog Read
5666	Keine Aktion
6666	Andere/unbekannte Aktion

Tabelle 72 - Codes für auszuführende Aktionen

Jeder Code steht für eine gleichnamige Operation, die ein Arduino Microcontroller auf einem analogen oder digitalen I/O-Pin ausführen kann (digitales bzw. analoges Lesen oder Schreiben), oder dient als Platzhalter.

Code	Datentyp	Kürzel
1111	Integer	INT
1112	Character	CHAR
1113	Boolean	BOOL
1114	HIGH/LOW	HILO
1115	Unsigned Character	UCHAR
1116	Unsigned Integer	UINT
1117	Long	LONG
1118	Unsigned Long	ULONG
1119	Float	FLOAT
1120	Double	DOUBLE
1121	String	STRING
1122	Byte	BYTE
9999	Unbekannt/Kein Typ	-

Tabelle 73 - Codes für Datentypen

5.8.6 Instruktion hochladen

Um eine Instruktion für einen Client auf einen ORCHID Server zu laden, übermittelt eine Drittsoftware einen speziell geformten PUT-Request. Die benötigten Header-Felder und der Aufbau dieses Requests können der folgenden Tabelle entnommen werden.

Header-Feld	Übertragener Wert
Board-ID	GUID des Clients, für den die Instruktion bestimmt ist
Component-ID	GUID des Aktors, für den die Instruktion bestimmt ist.
User-Agent	Zeichenkette, die den Kommunikationspartner als ORCHID Client oder mit dem Toolkit kompatible Drittsoftware identifiziert.
Host	Die IP-Adresse des Servers und der Port, auf dem sich die Drittsoftware verbinden möchte.
Authorization	Der Developer Key, der verwendet werden soll.
Action	Als Ganzzahl kodierte, auszuführende Aktion (vgl. Tabelle 72).
Value	In der Instruktion zu verwendender Wert.
Value-Type	Als Ganzzahl kodierter Datentyp von `Value` (vgl. Tabelle 73).
Posted	Zeitstempel (YYYY-MM-DD hh:mm:ss)
Priority	Prioritätswert der Instruktion.
Done	Boolescher Wert, der angibt, ob die Instruktion bereits bearbeitet wurde.
Pin	Der Pin, an dem die Instruktion durchgeführt werden soll.

Allgemeine Form

```
PUT HTTP/1.1
Board-ID:       <GUID>
Component-ID:   <GUID>
User-Agent:     <STRING>
Host:           <IP>:<PORT>
Authorization:  <DEVELOPER KEY>
Action:         <INT>
Value:          <STRING>
Value-Type:     <INT>
Posted:         <TIMESTAMP>
Priority:       <INT>
Done:           <BOOL>
Pin:            <INT>
Connection:     close
```

Beispiel

```
PUT HTTP/1.1
Board-ID: 9f476a50-d4e3-11e0-801d-001d92396635
Component-ID: 17e8a812-d574-11e0-a3c1-001d92396635
User-Agent: Orchid Arduino Toolkit
Host: 192.168.1.123:8181
Authorization: cd0887447e04c15b5128629f04b53c4c4858bb1e8ff2e8b6bfcb32dc99fb4e7198d72ed2
Action: 1666
Value: HIGH
Value-Type: 1114
Posted: 2011-08-19 16:25:45
Priority: 1
Done: false
Pin: 6
Connection: close
```

Tabelle 74 - Protokoll: Instruktionen hochladen

Das oben gezeigte Beispiel führt zur Speicherung einer Instruktion für einen Aktor mit der GUID *17e8a812-d574-11e0-a3c1-001d92396635* (angeschlossen an einen Client, der durch *9f476a50-d4e3-11e0-801d-001d92396635* identifiziert ist), die den Wert *HIGH* auf den digitalen I/O-Pin 6 schreibt.

Die Angabe eines Pins ist nicht zwingend notwendig (es sei denn, die Instruktion soll direkt von der Client-Bibliothek ausgeführt werden).

5.8.7 Sensordaten abrufen

Eine Drittsoftware kann einen GET-Request an einen ORCHID Server senden, um Sensordaten, die dieser für einen bestimmten Sensor gespeichert hat, abzufragen. Header-Felder und Aufbau eines solchen Requests werden in Tabelle 75 beschrieben.

Header-Feld	Übertragener Wert
User-Agent	Zeichenkette, die den Kommunikationspartner als ORCHID Client oder mit dem Toolkit kompatible Drittsoftware identifiziert.
Host	Die IP-Adresse des Servers und der Port, auf dem sich die Drittsoftware verbinden möchte.
Authorization	Der Developer Key, der verwendet werden soll.
Limit	Die maximale Anzahl von Datensätzen (optional)

Allgemeine Form

```
GET /<GUID>/<GUID> HTTP/1.1
User-Agent:     <STRING>
Host:           <IP>:<PORT>
Authorization:  <DEVELOPER KEY>
Limit:          <INT>
Connection:     close
```

Beispiel

```
GET /9f476a50-d4e3-11e0-801d-001d92396635/f6f9a4c0-dae6-4fcc-a1d2-5db57feac465 HTTP/1.1
User-Agent: Orchid Arduino Toolkit
Host: 192.168.1.123:8181
Authorization: cd0887447e04c15b5128629f04b53c4c4858bb1e8ff2e8b6bfcb32dc99fb4e7198d72ed2
Limit: 10
Connection: close
```

Tabelle 75 - Protokoll: Sensordaten abrufen

Der Server verhält sich bei der Verarbeitung solcher Anfragen ähnlich wie bereits in 5.8.5 beschrieben; die Pfadangabe nach dem Methodenkennzeichen GET enthält die GUID des Clients (erster Wert) und des Sensors (zweiter Wert), für den Daten abgerufen werden sollen.

Über das Header-Feld *Limit* wird bestimmt, wie viele Datensätze (maximal) angefordert werden sollen. Soll nur ein Datensatz abgerufen werden (dieser entspricht dem aktuellsten Sensorwert), kann das Feld ausgelassen oder in ihm der Wert 1 übermittelt werden. Um alle Sensordaten abzurufen, die ein Server jemals für die Kombination aus Client und Sensor gespeichert hat, wird als Limit der Wert 0 übertragen.

Können für die angegebenen GUIDs keine Sensordaten ermittelt werden, antwortet der Server mit HTTP-Code 404 (*„Not Found“*). Liegen Daten vor, wird eine Antwort mit HTTP-Code 200 („OK“)

übermittelt, die die aufbereiteten Sensordaten im Message Body enthält, wie das folgende Listing zeigt (Zeilenumbrüche wurden zur Verbesserung der Lesbarkeit hinzugefügt).

```
01 HTTP/1.1 200 OK
02 Connection: close
03 Content-Length: 199
04
05 sensor=f6f9a4c0-dae6-4fcc-a1d2-5db57feac465&
06 type=INT;INT&
07 value=972;123&
08 stamp=17:25:14 02.09.2011;17:25:16 02.09.2011
```

Listing 35 - Übermittlung von Sensordaten (Server-Antwort)

Die ermittelten Sensordaten werden als Liste von Schlüssel-Wert-Paaren übertragen, wobei die Felder *type* (Datentyp), *value* (Wert) und *stamp* (Zeitstempel) wiederum Listen als Wert enthalten. Der ORCHID Server garantiert, dass die Indizes von Werten eines Datensatzes in allen Listen stets konsistent sind. Diese Garantie wird im obigen Listing durch die farbliche Hervorhebung der Werte verdeutlicht (Werte gleicher Farbe gehören zum gleichen Datensatz).

Für den jeweiligen Datentyp der ermittelten Werte wird das Datentyp-Kürzel (vgl. 5.8.5) übermittelt.

5.8.8 Client-Informationen abrufen

Um ausführliche Informationen über einen Client von einem ORCHID Server abzurufen, kann eine Drittsoftware einen SEARCH-Request an den Server senden, dessen benötigte Header-Felder und Aufbau der folgenden Tabelle entnommen werden können.

Header-Feld	Übertragener Wert
Board-ID	Die GUID des Clients, dessen Informationen abgerufen werden sollen.
User-Agent	Zeichenkette, die den Kommunikationspartner als ORCHID Client oder mit dem Toolkit kompatible Drittsoftware identifiziert.
Host	Die IP-Adresse des Servers und der Port, auf dem sich die Drittsoftware verbinden möchte.
Authorization	Der Developer Key, der verwendet werden soll.
Allgemeine Form	

<pre>SEARCH HTTP/1.1
Board-ID: <GUID>
User-Agent: <STRING>
Host: <IP>:<PORT>
Authorization: <DEVELOPER KEY>
Connection: close</pre>	

Beispiel	

<pre>SEARCH HTTP/1.1
Board-ID: 9f476a50-d4e3-11e0-801d-001d92396635
User-Agent: Orchid Arduino Toolkit
Host: 192.168.1.123:8181
Authorization: cd0887447e04c15b5128629f04b53c4c4858bb1e8ff2e8b6bfcb32dc99fb4e7198d72ed2
Connection: close</pre>	

Tabelle 76 - Protokoll: Client-Informationen abrufen

Kann der Server die gewünschten Informationen nicht abrufen, wird eine Fehlermeldung mit HTTP-Code 404 (*"Not Found"*) als Antwort gesendet, ansonsten werden die ermittelten Informationen entsprechend dem im folgenden Listing gezeigten Schema übermittelt (Zeilenumbrüche wurden zur Verbesserung der Lesbarkeit hinzugefügt).

```
01 HTTP/1.1 200 OK
02 Connection: close
03 Content-Length: 433
04
05 arduino=fdb41088-6cf5-4bd8-b71e-4e40e292d7f0&
   name=Test-Arduino&
   description=Ein Arduino zum Testen&
   latitude=52550506&
   longitude=13349386&
   firstseen=14.05.2011 22:49:41&
   lastseen=02.09.2011 17:49:21&
   lastip=192.168.1.145&
   sensors=531a3bbe-2bec-4bff-a4c3-c60645943f6d&
   components=fec2b463-600a-45a7-9494-e844b03f7257;
              abc2b463-600a-45a7-9494-e844b03f7257;
              49c77f2b-0bd9-49a5-a940-50ff8bb2d9c3&
   model=Arduino Duemilanove with Ethernet Shield
```

Listing 36 - Übermittlung von Client-Informationen (Server-Antwort)

Die Client-Informationen werden auch in dieser Antwort als Schlüssel-Wert-Paare übermittelt, die eine Drittsoftware entsprechend verarbeiten können muss. Für jeden Sensor und Aktor, der an den Client angeschlossen ist, wird die GUID übertragen; mit dieser Information kann eine Drittsoftware weitere Informationen abrufen oder Instruktionen hochladen, wenn dies gewünscht ist.

Alle weiteren Datenfelder entsprechen im Wesentlichen der Struktur der Datenbanktabelle, in der Informationen über Clients gespeichert werden (vgl. 5.10).

5.9 Abläufe und Verhaltensweisen

In diesem Abschnitt werden ausgewählte Programmabläufe und Verhaltensweisen der Server-Komponente im Bezug auf die Behandlung und Beantwortung von HTTP-Anfragen durch Clients und Drittsoftware beschrieben.

Es handelt sich dabei ausdrücklich nicht um eine vollständige Beschreibung aller Programmabläufe innerhalb der Server-Komponente.

5.9.1 Multithreading

Die Server-Komponente des Toolkits ist darauf ausgelegt, Requests von Clients möglichst schnell zu behandeln, so dass weitere eingehende Anfragen nicht abgelehnt werden müssen bzw. der Server weiterhin erreichbar bleibt. Dadurch allein kann jedoch nicht garantiert werden, dass weitere Anfragen nicht dennoch erst nach einer Wartezeit bearbeitet oder sogar abgelehnt werden.

Um dieses Problem zu beseitigen, nutzt die Server-Komponente *Threads (dt.: „Fäden“)*, in denen die einzelnen Anfragen von Clients abgearbeitet werden. So muss für jeden eingehenden Request lediglich bestimmt werden, welche Instanz diesen bearbeiten soll und ein Thread gestartet werden, in dem die Bearbeitung abläuft.

5.9.1.1 Threads in C#

Die Verwendung von Threads in C# erfordert die Einbindung der Klasse `Thread`. Diese Klasse stellt verschiedene Methoden zur Nutzung von Threads zur Verfügung, an dieser Stelle soll jedoch nur die Methode `Start()` betrachtet werden, die einen Thread startet, nachdem dieser initialisiert wurde. Bei der Initialisierung muss dem zu startenden Thread u.a. mitgeteilt werden, welche Methode er während seiner Laufzeit ausführen soll. Diese Methode wird „*ThreadStart-Methode*“ genannt und der Thread läuft so lange, bis diese Methode beendet ist oder ein anderes Ereignis eintritt, das zu dessen Beendigung führt (z.B. ein Timeout oder ein Fehler). Der Name der Methode basiert auf dem in der Klasse `Thread` enthaltenen *Delegate (dt.: „Abgeordneter“* oder *„Beauftragter“*, hier aber auch: *„Mittelsmann“*) vom Typ `ThreadStart`. Die Übersetzung mit „Mittelsmann“ erscheint deshalb sinnvoll, da beim Aufruft von `Thread.Start()` auch die Methode aufgerufen wird, auf die der Delegate verweist, er also als Vermittler zwischen aufrufendem und neu gestartetem Thread auftritt.

Der Lebenszyklus eines Threads kann wie in der folgenden Grafik veranschaulicht werden:

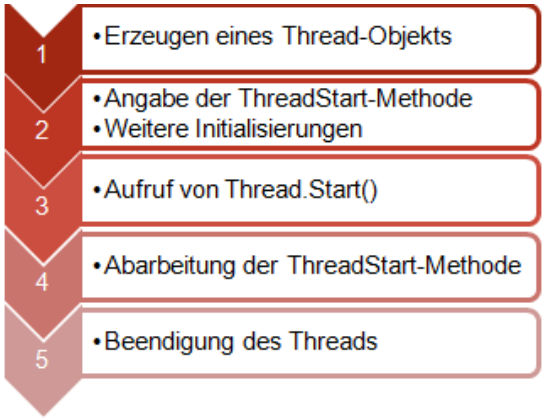

Abbildung 23 - Lebenszyklus eines Threads

Wie bereits erwähnt kann es vorkommen, dass die ThreadStart-Methode nicht vollständig abgearbeitet werden kann, da der Thread zuvor durch ein bestimmtes Ereignis wie einen Timeout oder einen Fehler unterbrochen wird.

Durch die Verwendung von Threads zur Abarbeitung einer bestimmten Methode (der ThreadStart-Methode) ergibt sich der Vorteil, dass der startende Thread lediglich die in Abbildung 23 gezeigten Schritte 1 bis 3 ausführen muss. Der eigentliche, rechenintensive Teil der Bearbeitung, nämlich die ThreadStart-Methode, wird vom neu erzeugten Thread ausgeführt; der startende Thread kann währenddessen bereits andere Aufgaben übernehmen.

Jedes Programm, das in C# unter Mono oder dem .NET Framework entwickelt wird, wird zur Laufzeit durch mindestens einen Thread, den *Main Thread (dt. etwa: „Hauptfaden“)*, repräsentiert. Dieser Thread muss vom Entwickler nicht explizit angelegt werden, sondern wird durch die ausführende virtuelle Maschine (CLR bzw. Mono Runtime, vgl. 2.3.1) erzeugt. Weitere Threads werden – für den Entwickler nicht sichtbar – ebenfalls von der virtuellen Maschine erzeugt, z.B. für die Ausführung von Code innerhalb einer Bibliothek.

5.9.1.2 Multithreading in der Server-Komponente

Beim Start des Servers wird (im *Main Thread* der Anwendung) eine Instanz der Klasse `Bloom` erzeugt. Dieser Instanz werden im Konstruktor zwei Fabrik-Instanzen übergeben: `ServerFactory` und `DaemonFactory` (vgl. 5.3.1). Mit diesen Fabriken werden im weiteren Verlauf Instanzen von Server- und Daemon-Klassen erzeugt. Die Klasse `Bloom` bietet nur die öffentliche Methode `Run()` an. Diese Methode erzeugt über die Fabrikklassen je eine Instanz der Klassen `OrchidHardwareServer` und `OrchidSoftwareServer` und übergibt deren Methode `Start()` als ThreadStart-Methode an jeweils einen eigenen Thread. Jede Server-Instanz läuft also in einem eigenen Thread, der die Methode `Start()` des jeweiligen Servers abarbeitet.

Die `Start()`-Methode der Server-Klassen beinhaltet eine Schleife – sie wird erst dann beendet, wenn diese nicht mehr wiederholt oder der entsprechende Thread auf andere Weise beendet oder abgebrochen wird. Innerhalb der Schleife wartet jede der beiden Server-Instanzen an einem ihr zugewiesenen Port auf eine eingehende Verbindung. Würde dieses Warten nicht in einem eigenen, sondern im *Main Thread* der Anwendung durchgeführt, wäre sie so lange blockiert, bis eine wartende Server-Instanz eine eingehende Anfrage erhält. Anfragen an die zweite Server-Instanz würden während dieser Blockierung nicht behandelt werden.

Trifft eine Anfrage an einem von einer Server-Instanz überwachten Port ein, erzeugt diese über die `DaemonFactory` eine Instanz einer Daemon-Klasse (je nach Server-Klasse entweder vom Typ `OrchidHardwareDaemon` oder `OrchidSoftwareDaemon`) und übergibt deren Methode `HandleRequest()` als ThreadStart-Methode an einen neuen Thread. Innerhalb dieses weiteren Threads (*„Daemon-Thread“*) wird die Anfrage behandelt, während der Server-Thread bereits wieder den ihm zugewiesenen Port beobachten kann.

Abbildung 24 bildet den hier geschilderten Ablauf in einem UML-Sequenzdiagramm (für eine Instanz von `OrchidHardwareServer`) ab.

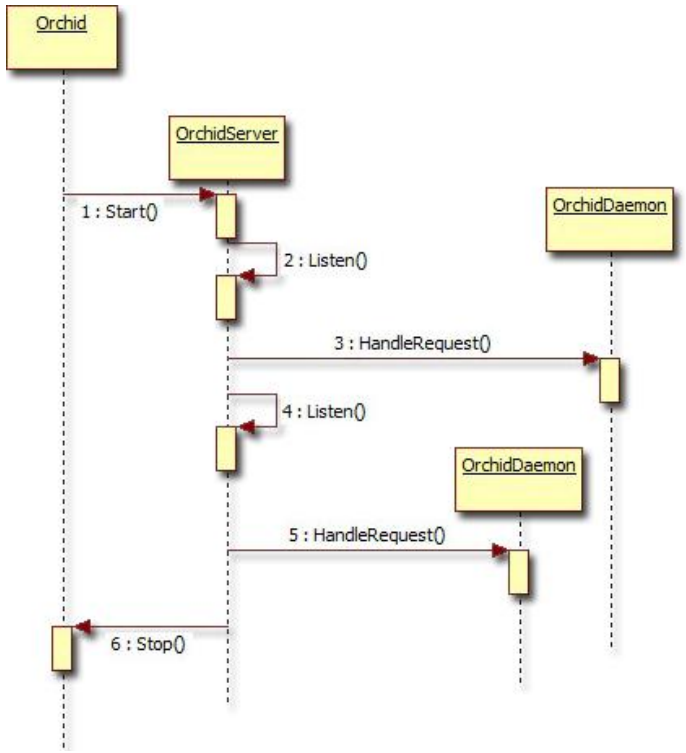

Abbildung 24 - Multithreading (OrchidHardwareServer)

5.9.2 Behandlung von Anfragen in der Server-Komponente

Wie bereits im vorhergehenden Abschnitt beschrieben, setzt die Server-Komponente Threads ein, um Anfragen von Clients zu behandeln, ohne die Ausführung der Software zu unterbrechen zu müssen. Dieser Abschnitt beschreibt detailliert die Abläufe, die durch einen Aufruf der Methode `HandleRequest()` einer Instanz von `OrchidHardwareDaemon` bzw. `OrchidSoftwareDaemon` durch eine entsprechende Server-Instanz ausgelöst werden.

Allgemein betrachtet setzt der Aufruf der Methode `HandleRequest()` einer Daemon-Instanz stets die folgenden Prozesse in Gang:

1. Identifikation des Verbindungspartners und dessen Speicherung
2. Auslesen und Transformieren der Anfrage-Daten (*„Parsen“*)
3. Bestimmen der Art der Anfrage, basierend auf der HTTP-Methode
4. Aufruf derjenigen Methode, die die bestimmte Art von Anfrage behandelt

In den folgenden Abschnitten werden Anfragen, die eine bestimmte HTTP-Methode verwenden, als entsprechende *Requests* bezeichnet. Die Methode eines HTTP-Requests wird in dessen erstem Header-Feld kodiert (vgl. 2.2). Eine Anfrage, die *HTTP-GET* nutzt, wird also im Folgenden als *GET-Request* bezeichnet.

5.9.2.1 Allgemeines zur Behandlung von Anfragen

Jede Art von Request steht für eine bestimmte Aktion, die ein Verbindungspartner auf dem Server auslösen möchte. Entsprechend werden bei jeder Anfrage unterschiedliche Daten benötigt, die als Header-Felder übermittelt werden müssen (vgl. 5.8). Instanzen von `OrchidHardwareDaemon` und `OrchidSoftwareDaemon` sind in der Lage, Anfragen bezüglich des Vorhandenseins dieser benötigten Header-Felder zu überprüfen. Ist ein Request aufgrund fehlender Header-Felder ungültig, wird eine entsprechende Antwort an den Verbindungspartner übermittelt (HTTP-Code 400, *„Bad Request“*) und die Verarbeitung abgebrochen.

Das Header-Feld *„User-Agent“* wird gesondert betrachtet (dieses Verhalten kann über die Server-Konfiguration ein- und ausgeschaltet werden). In diesem Feld wird die Identifikation eines Verbindungspartners übermittelt. Die Konfiguration des Servers ermöglicht es, nur Anfragen zu bearbeiten, die in diesem Feld einen festgelegten Wert übertragen. Der Wert, der in der Referenzimplementierung verwendet wird, lautet *„Orchid Arduino Toolkit“*. Er kann jederzeit über die Konfigurationsdatei verändert werden.

Tritt während der Verarbeitung eines gültigen Requests ein Fehler auf, wird die Verarbeitung abgebrochen und dem Verbindungspartner eine Fehlermeldung übermittelt (HTTP-Code 500, *„Internal Server Error“*).

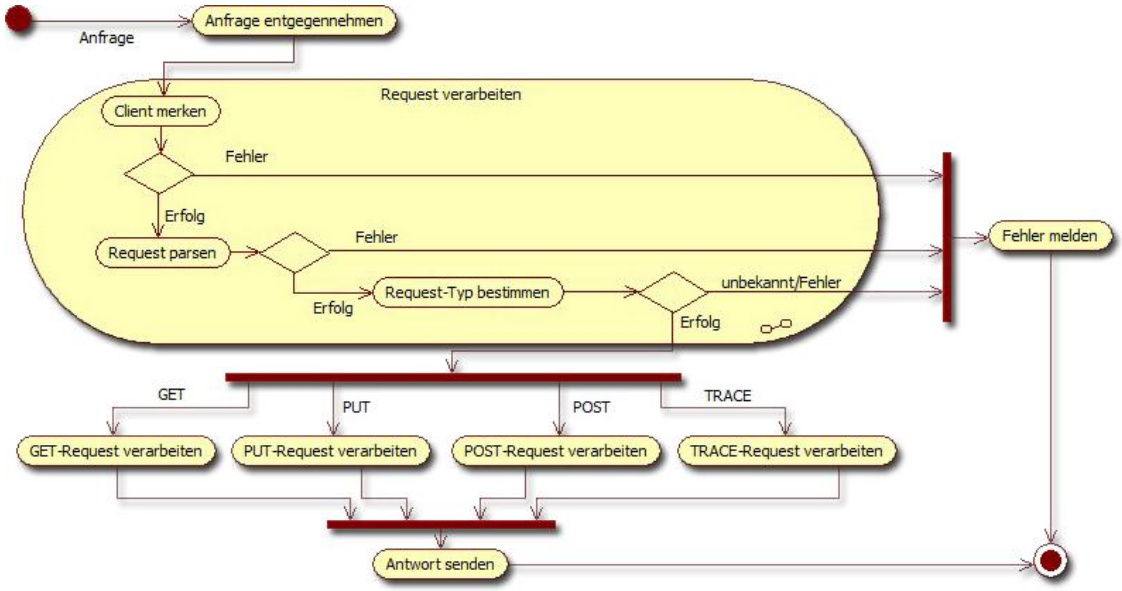

Abbildung 25 - Allgemeine Bearbeitung von Requests

Die obige Abbildung 25 zeigt in einem UML-Aktivitätsdiagramm auszugsweise, wie die Server-Komponente Anfragen allgemein bearbeitet. Grundsätzlich ist der Ablauf für jede Art von Request der gleiche. Zunächst wird der Verbindungspartner (im Diagramm: *„Client“*) gespeichert, daraufhin die Anfrage geparst und ihr Typ bestimmt. Ist dieser bekannt, wird sie an der entsprechenden Stelle verarbeitet und beantwortet (die Antwort variiert je nach Anfrage und Bearbeitungserfolg).

Die in diesem Abschnitt enthaltenen Abbildungen zeigen die Schritte zur Authentifizierung und Berechtigungsprüfung nicht (vgl. 6.3.3.2). Diese Prüfungen werden in allen Methoden durchgeführt, die eine Anfrage behandeln, wurden jedoch aus Gründen der Übersichtlichkeit in den Diagrammen ausgespart.

5.9.2.2 Client-GET-Requests

Ein GET-Request, der von einer Instanz von `OrchidHardwareServer` entgegengenommen und von einer Instanz von `OrchidHardwareDaemon` verarbeitet wird, steht für eine Anfrage eines Clients, ob Instruktionen für eine an diesen Client angeschlossene Komponente (einen Aktor) vorliegen. Die Abarbeitung eines GET-Requests geschieht in der Methode `HandleGETRequest()` der Klasse `OrchidHardwareDaemon`.

Nachdem sichergestellt wurde, dass die zur Verarbeitung des GET-Requests notwendigen Daten vorliegen, wird zunächst der übermittelte Developer Key geprüft. Ist der Key nicht gültig, wird die Verarbeitung abgebrochen und eine Fehlermeldung an den Client gesendet (HTTP-Code 401, *„Unauthorized“*). Im nächsten Schritt werden die ID des Clients und der Komponente aus dem

Request ausgelesen. Tritt dabei ein Fehler auf (z.B. weil eine ID ungültig ist), wird eine Fehlermeldung (HTTP-Code 400, *„Bad Request“*) an den Client gesendet und die Verarbeitung abgebrochen.

Liegen die IDs von Client und Komponente vor, wird geprüft, ob es diesem Client erlaubt ist, Instruktionen vom Server abzurufen. Dazu werden über die Methode `GetCredentials(Guid)` der Klasse `MySqlAdapter` die Berechtigungen des Clients aus der Datenbank ausgelesen. Ist es dem Client nicht erlaubt, Instruktionen abzurufen, wird die Anfrage mit einer Fehlermeldung (HTTP-Code 401, *„Unauthorized“*) beantwortet und die Verarbeitung beendet.

Um die nächste vorliegende Instruktion abzurufen, wird die Methode `GetNextInstruction(Guid, Guid)` der Klasse `MySqlAdapter` aufgerufen. Diese sucht in der Datenbank nach der nächsten, bisher unbehandelten Instruktion für die Kombination aus Client und Aktor. Diese Instruktion wird zurückgegeben (oder `null`, falls keine solche Instruktion gefunden wurde). Wurde keine Instruktion gefunden, wird dem Client eine entsprechende Antwort gesendet (HTTP-Code 404, *„Not found“*) und die Verarbeitung des GET-Requests beendet. Liegt eine Instruktion vor, wird sie für eine Antwort an den Client aufbereitet und dann an diesen gesendet (HTTP-Code 200, *„OK“*) und die Verarbeitung der Anfrage beendet.

Abbildung 26 zeigt den geschilderten Programmablauf in einem UML-Aktivitätsdiagramm.

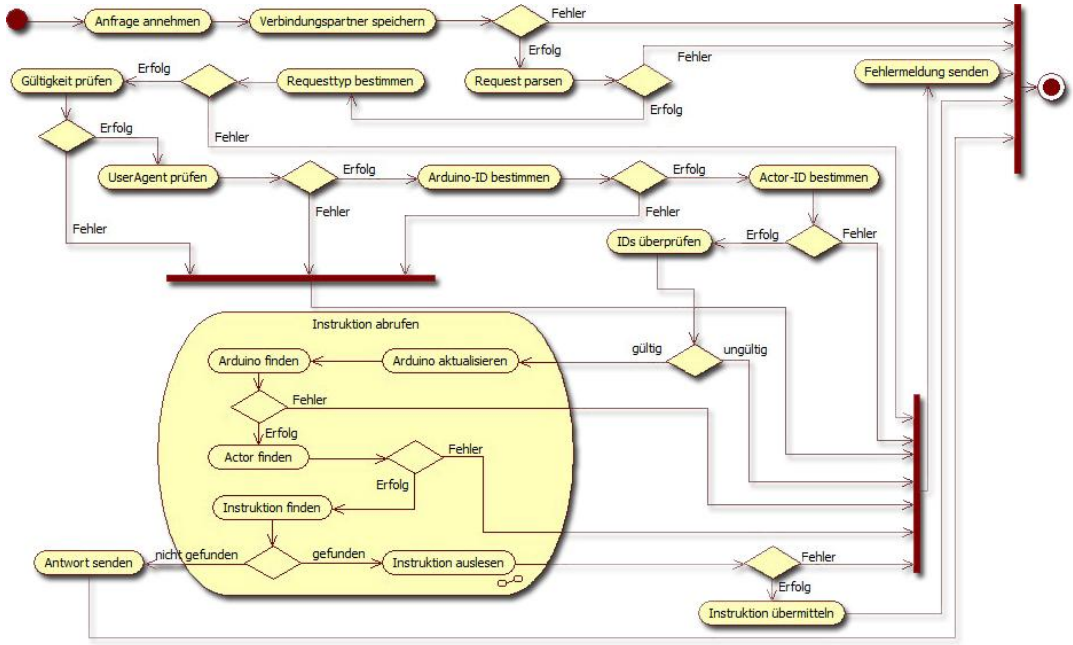

Abbildung 26 - Bearbeitung von Client-GET-Requests

5.9.2.3 Client-PUT-Requests

Ein PUT-Request, der von einer Instanz von `OrchidHardwareServer` entgegengenommen und von einer Instanz von `OrchidHardwareDaemon` verarbeitet wird, steht für eine Anfrage eines Clients, Sensordaten auf dem Server zu speichern. Die Verarbeitung eines solchen Requests geschieht in der Methode `HandlePUTRequest` in der Klasse `OrchidHardwareDaemon`.

Handelt es sich um einen gültigen PUT-Request, werden zunächst die ID des Clients und des Sensors aus der Anfrage ausgelesen. Nur wenn beide IDs erfolgreich ermittelt werden können, wird die Verarbeitung fortgesetzt; ansonsten wird sie an dieser Stelle abgebrochen und eine Fehlermeldung an den Client übermittelt (HTTP-Code 400, *„Bad Request“*).

Sind die IDs ermittelt worden, wird die Integrität der übermittelten Daten verifiziert (dieses Verhalten kann über die Konfiguration abgeschaltet werden). Können die Daten nicht eindeutig einem Datentyp zugeordnet werden oder ist ein Wert für den angegebenen Datentyp nicht gültig (z.B. bei Übermittlung von *„ungültig“* als Integer-Wert), wird eine entsprechende Meldung an den Client gesendet (HTTP-Code 400, *„Bad Request“*) und die Verarbeitung abgebrochen.

Die bisher aus dem PUT-Request ermittelten Daten (Arduino-ID, Sensor-ID, Datentyp und Wert) werden nun an die Methode `StoreSensorData(Guid, Guid, string, string)` der Klasse `MySqlAdapter` übergeben, die zur Speicherung von Sensordaten in der Datenbank verwendet wird.

Die Methode prüft zunächst, ob der übermittelte Sensor laut Datenbank an den angegebenen Client angeschlossen ist. So soll sichergestellt werden, dass keine Dateninkonsistenzen auftreten, weil ein Client irrtümlich eine falsche ID für einen Sensor übermittelt. Ist diese Prüfung erfolgreich, werden die übergebenen Daten in der Datenbank gespeichert, ansonsten wird ein Fehlercode zurückgegeben (ein Wert vom Typ `OrchidDataResult`).

Ist die Rückgabe der Methode `StoreSensorData(Guid, Guid, string, string)` gleich dem Wert `OrchidDataResult.Inserted`, wurden die übermittelten Daten erfolgreich gespeichert und es wird eine Erfolgsmeldung an den Client gesendet (HTTP-Code 200, *„OK“*). Ansonsten trat ein Fehler auf, der ebenfalls an den Client übermittelt wird (der Code variiert je nach Fehler).

Abbildung 27 zeigt den geschilderten Programmablauf in einem UML-Aktivitätsdiagramm.

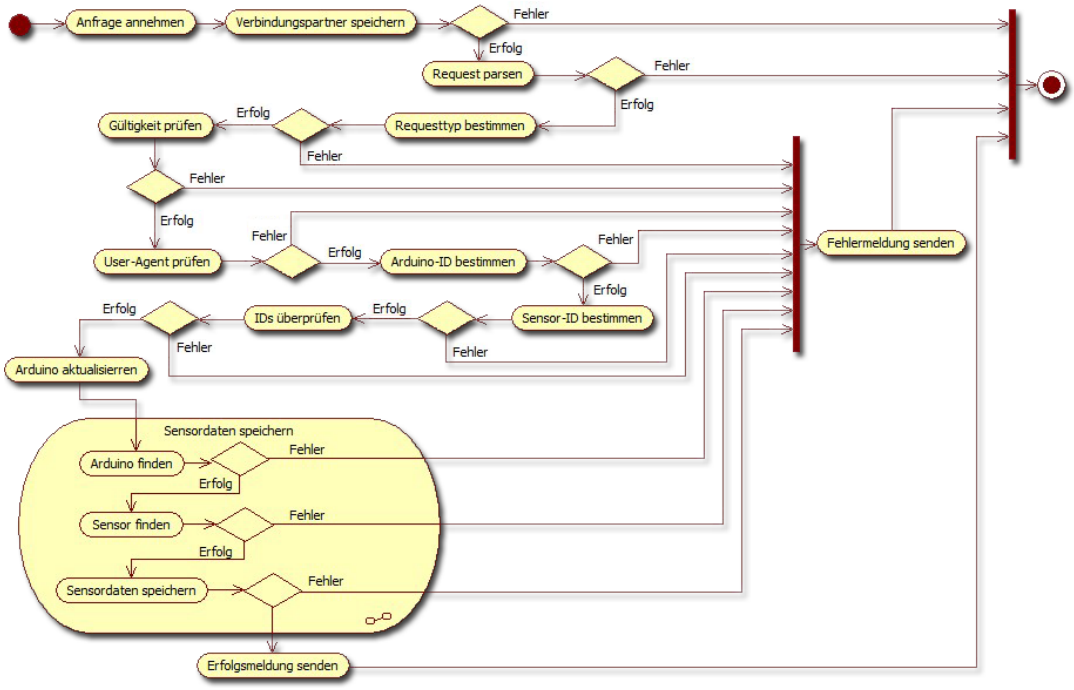

Abbildung 27 - Bearbeitung von Client-PUT-Requests

5.9.2.4 Client-TRACE-Requests

TRACE-Requests dienen einem Client dazu, dem Server mitzuteilen, dass er bereit ist. Sie werden im Umfeld dieser Arbeit auch als *„Heartbeat-Requests“ (deutsch: „Herzschlag-Anfragen“)* bzw. lediglich als *„Heartbeats“* bezeichnet.

Zur Behandlung eines Heartbeat-Requests wird die Methode `HandleTRACERequest()` der Klasse `OrchidHardwareDaemon` aufgerufen. In ihr wird zunächst die ID des Clients ausgelesen, der den

Heartbeat übermittelt hat. Ist die ID ungültig oder wurde keine ID angegeben, wird eine Fehlermeldung an den Client gesendet (HTTP-Code 400, *„Bad Request“*) und die Verarbeitung der Anfrage abgebrochen.

Liegen alle notwendigen Daten vor und sind diese verifiziert, wird die Methode `TouchArduino(AArduino, string)` aus der Klasse `MySqlAdapter` aufgerufen. Diese Methode aktualisiert den Datensatz eines Arduinos in der Datenbank mit einem aktuellen Zeitstempel und der übergebenen IP-Adresse.

Ist die Aktualisierung erfolgreich, wird eine Bestätigungsmeldung an den Client gesendet (HTTP-Code 200, *„OK“*), ansonsten eine Fehlermeldung (HTTP-Code 400, *„Bad Request“* oder HTTP-Code 500, *„Internal Server Error“*).

Abbildung 28 zeigt den geschilderten Programmablauf in einem UML-Aktivitätsdiagramm.

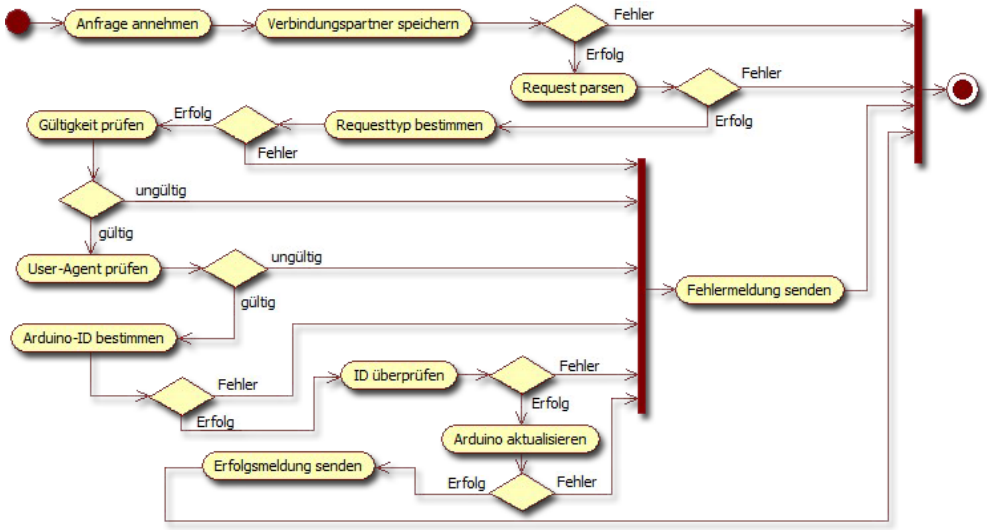

Abbildung 28 - Bearbeitung von Client-TRACE-Requests

5.9.2.5 Drittsoftware-GET-Requests

GET-Requests werden von einer Software eingesetzt, um auf dem Server vorhandene Sensordaten abzurufen. Diese Art von Anfragen wird von der Methode `HandleGETRequest()` der Klasse `OrchidSoftwareDaemon` verarbeitet.

Zunächst werden die IDs des Clients und des Sensors, für deren Kombination Sensordaten abgerufen werden sollen, aus dem Request ausgelesen. Ist dies nicht möglich oder tritt dabei ein Fehler auf, ist die Anfrage ungültig und ihre Verarbeitung wird mit einer Fehlermeldung an den Verbindungspartner abgebrochen (HTTP-Code 400, *„Bad Request“*).

Liegen die IDs für Client und Sensor vor, wird die Methode `GetSensorData(Guid, Guid, int, bool)` der Klasse `MySqlAdapter` aufgerufen. Diese Methode prüft, ob der übermittelte Sensor tatsächlich an den angegebenen Client angeschlossen ist (so wird verhindert, dass irrtümlich Daten für einen anderen als den gewünschten Sensor ausgelesen werden). Ist dies nicht der Fall, wird die Verarbeitung des Requests mit einer Fehlermeldung an den Verbindungspartner (HTTP-Code 400, *„Bad Request“*) abgebrochen. Ist der Sensor an den angegebenen Client angeschlossen, werden die gewünschten Sensordaten aus der Datenbank ausgelesen und an den Verbindungspartner übermittelt (HTTP-Code 200, *„OK“*).

Fragt ein Verbindungspartner nach einem einzelnen Datensatz (dem aktuellsten Datensatz, der für den angegebenen Sensor gespeichert wurde), wird eine Antwort wie die folgende übermittelt:

```
01 sensor=531a3bbe-2bec-4bff-a4c3-c60645943f6d&type=INT&value=12&stamp=3:28:21 PM
   8/21/2011
```

Listing 37 - Übermittlung eines einzelnen Sensorwerts (Server-Antwort)

Das Feld *sensor* enthält die GUID des betroffenen Sensors, *type* gibt an, welche Art von Daten ermittelt wurden, *value* repräsentiert den eigentlichen Wert dieser Daten und *stamp* steht für den Zeitpunkt, zu dem die Daten gespeichert wurden.

Fragt ein Verbindungspartner mehr als einen Sensorwert ab, verändert sich die Antwort entsprechend (zur Verbesserung der Lesbarkeit wurden Zeilenumbrüche hinzugefügt):

```
01 sensor=531a3bbe-2bec-4bff-a4c3-c60645943f6d
   &type=INT;INT;INT;
   &value=123;862;863;
   &stamp=3:28:21 PM 8/21/2011;3:12:08 PM 8/21/2011;3:12:06 PM 8/21/2011;
```

Listing 38 - Übermittlung mehrerer Sensorwerte (Server-Antwort)

Diese Antwort enthält die gleichen Felder wie die vorherige; die übermittelten Werte für diese Felder bestehen nun jedoch aus einer Liste von Werten. Eine Software, die einen GET-Request an einen ORCHID Server stellt, muss diese Antwort entsprechend verarbeiten können.

Die folgende Abbildung 29 zeigt den geschilderten Programmablauf in einem UML-Aktivitätsdiagramm.

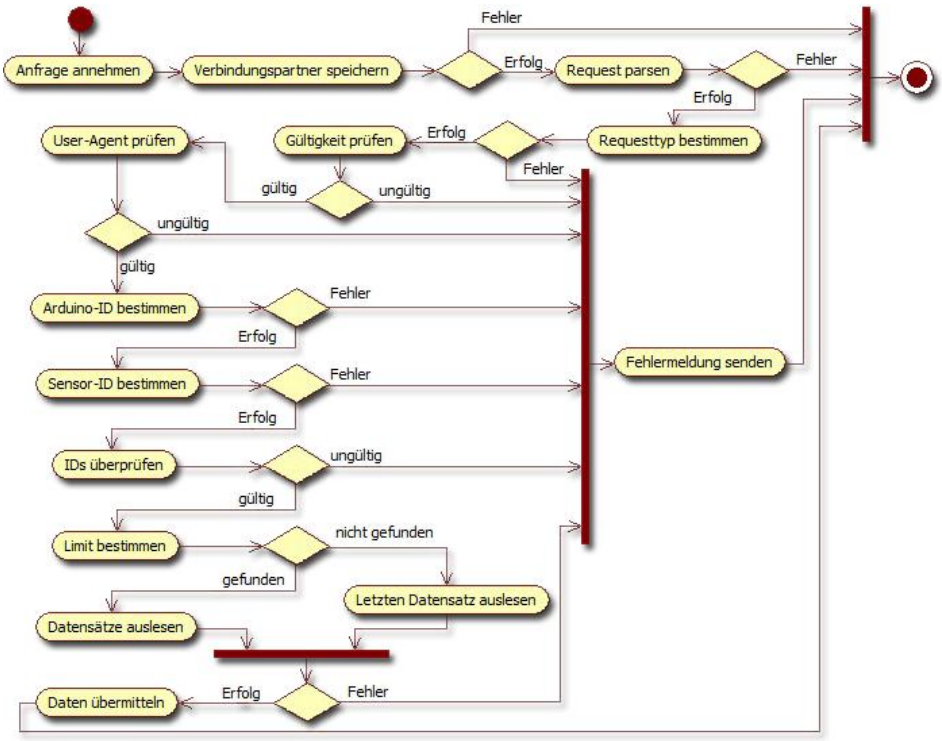

Abbildung 29 - Bearbeitung von Drittsoftware-GET-Requests

5.9.2.6 Drittsoftware-PUT-Request

Sendet eine Software einen PUT-Request an den Server, bedeutet das, dass sie eine Instruktion für einen Client hochladen möchte. Solche Requests werden in der Methode `HandlePUTRequest()` der Klasse `OrchidSoftwareDaemon` behandelt.

PUT-Requests dieser Art sind die komplexesten Requests, die die Server-Komponente behandeln muss, da eine Instruktion stets aus zahlreichen Datenfeldern besteht (vgl. 5.8.5 und 5.8.6). Zunächst wird die Gültigkeit des Requests geprüft. Kann diese nicht sichergestellt werden, oder tritt dabei ein Fehler auf, wird die Verarbeitung beendet und eine Fehlermeldung an die anfragende Software gesendet (HTTP-Code 400, *„Bad Request“*).

Ist der Request gültig, werden zuerst die IDs für den Client und den Aktor, für deren Kombination eine neue Instruktion auf dem Server hinterlegt werden soll, aus den Header-Feldern des Requests ausgelesen. Sind die IDs ungültig oder nicht lesbar, wird die Verarbeitung mit einer Fehlermeldung an den Verbindungspartner beendet (HTTP-Code 400, *„Bad Request“*). Sind die IDs gültig, werden weitere Werte aus dem Request ausgelesen, darunter u.a. die auszuführende Aktion, ein Zeitstempel, der I/O-Pin des Arduinos und die Priorität der Instruktion (vgl. 5.8.5 und 5.8.6).

Liegen alle benötigten Werte vor (optionale Felder werden, falls sie nicht vorhanden sind, mit Standardwerten gefüllt), wird eine neue Instanz der Klasse `Instruction` erzeugt und über die Methode `StoreInstruction(AInstruction)` der Klasse `MySqlAdapter` in der Datenbank gespeichert.

Diese Methode versucht zunächst, die übergebenen Werte in der Datenbank zu lokalisieren. Schlägt dieser Versuch fehl, wird dem Verbindungspartner eine Fehlermeldung übermittelt (HTTP-Code 400, *„Bad Request“*) und die Verarbeitung der Anfrage abgebrochen. Sind alle benötigten Werte vorhanden, wird die neue Instruktion der Datenbank gespeichert und dem Verbindungspartner eine Erfolgsmeldung gesendet (HTTP-Code 200, *„OK“*).

Abbildung 30 zeigt den geschilderten Programmablauf in einem UML-Aktivitätsdiagramm.

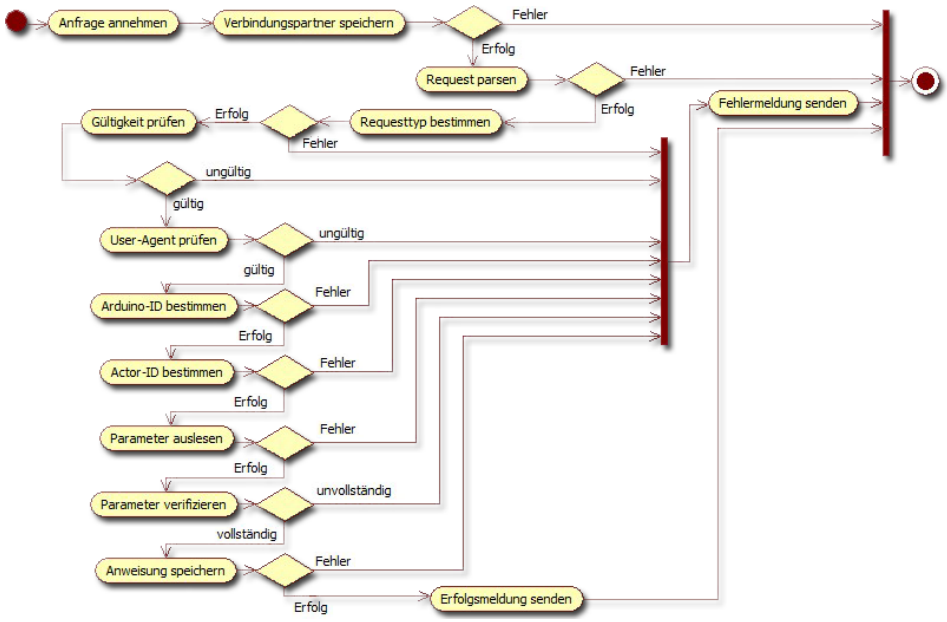

Abbildung 30 - Bearbeitung von Drittsoftware-PUT-Requests

5.9.2.7 Drittsoftware-SEARCH-Request

Über einen SEARCH-Request an den Server kann eine Software Informationen über einen Client und die an diesen Client angeschlossenen Komponenten abrufen. SEARCH-Requests werden von der Methode `HandleSEARCHRequest()` der Klasse `OrchidSoftwareDaemon` behandelt.

Wie bei allen anderen Request-Arten auch, wird zunächst die Gültigkeit der Anfrage überprüft und, falls sie nicht gültig ist oder ein Fehler bei der Verarbeitung auftritt, eine Fehlermeldung an den Verbindungspartner gesendet (HTTP-Code 400, *„Bad Request“*).

Ein gültiger SEARCH-Request enthält mindestens die GUID des Clients, über den Informationen abgerufen werden sollen. Diese ID wird nach der Gültigkeitsprüfung aus der Anfrage ermittelt und verifiziert. Handelt es sich um eine gültige ID, wird die Methode `GetArduinoInfo(Guid)` der Klasse `MySQLAdapter` aufgerufen, um die geforderten Informationen aus der Datenbank auszulesen.

Diese Methode versucht zunächst, den angegebenen Client in der Datenbank zu finden. Schlägt dieser Versuch fehl, wird die Verarbeitung der Anfrage mit einer Fehlermeldung an den Verbindungspartner abgebrochen (HTTP-Code 404, *„Not Found“*). Befindet sich der gesuchte Arduino in der Datenbank, werden zunächst dessen Daten gelesen und zwischengespeichert. Danach wird versucht, auch die an den Arduino angeschlossenen Sensoren und Aktoren in der Datenbank zu finden und deren GUIDs auszulesen. Werden keine Daten gefunden, führt dies nicht zu einem Fehler, da auch ein Arduino ohne angeschlossene Komponenten mit dem ORCHID Toolkit verwendet werden können soll.

Die folgende Abbildung 31 zeigt den beschriebenen Ablauf in einem UML-Aktivitätsdiagramm.

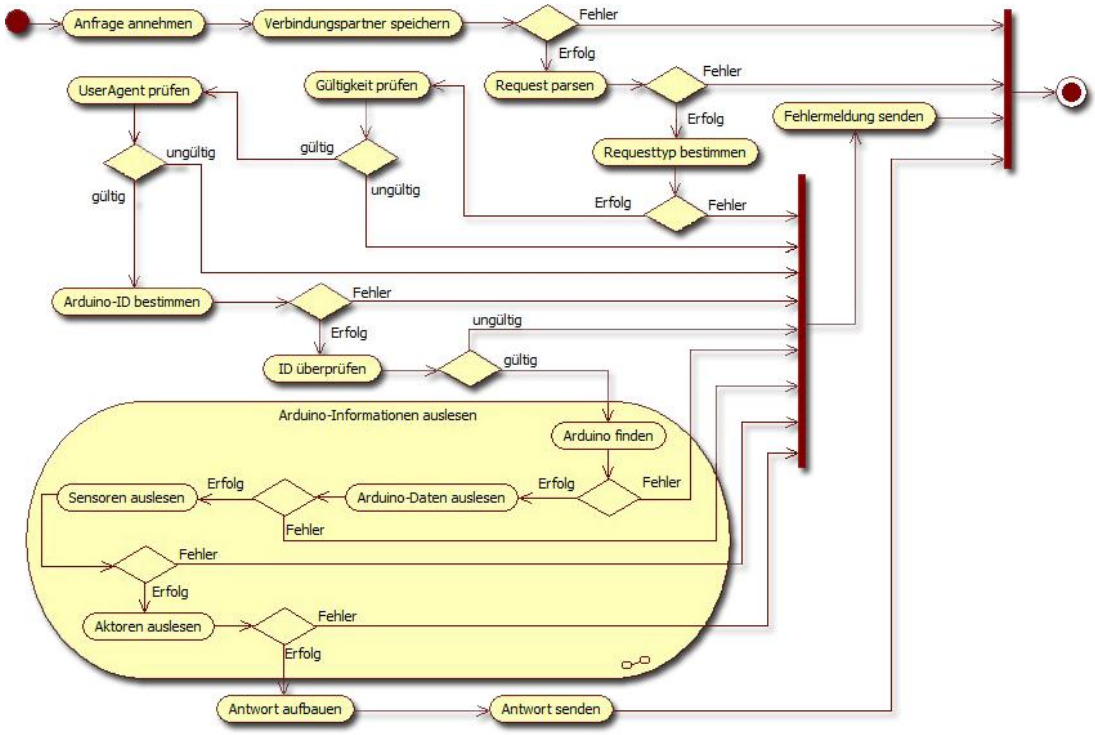

Abbildung 31 - Bearbeitung von Drittsoftware-SEARCH-Requests

Nachdem alle gesuchten Daten aus der Datenbank ausgelesen wurden, wird die Antwort an den Verbindungspartner zusammengesetzt und daraufhin übermittelt (HTTP-Code 200, *„OK“*). Eine solche Antwort hat stets die folgende Form (zur Verbesserung der Lesbarkeit wurden Zeilenumbrüche hinzugefügt):

```
arduino=fdb41088-6cf5-4bd8-b71e-4e40e292d7f0&
name=Test-Arduino&
description=Ein Arduino zum Testen&
latitude=52.550506&
longitude=13.349386&
firstseen=5/14/2011 10:49:41 PM&
lastseen=8/23/2011 6:31:49 PM&
lastip=192.168.1.141&
sensors=531a3bbe-2bec-4bff-a4c3-c60645943f6d&
components=fec2b463-600a-45a7-9494-e844b03f7257;
           abc2b463-600a-45a7-9494-e844b03f7257;
           49c77f2b-0bd9-49a5-a940-50ff8bb2d9c3
```

Listing 39 - Übermittlung von Client-Informationen (Server-Antwort)

Listing 39 zeigt eine beispielhafte Antwort auf einen SEARCH-Request. Eine solche Antwort enthält neben allgemeinen Informationen zum angeforderten Arduino (z.B. Name, Beschreibung und geografische Position) auch die GUIDs der angeschlossenen Sensoren (Feld `sensors`) und Aktoren (Feld `components`). Darüber hinaus wird zusätzlich mitgeteilt, wann der Arduino zum ersten und zum letzten Mal Verbindung mit dem Server aufgenommen (`firstseen` und `lastseen`), und welche IP-Adresse er zuletzt verwendet hat.

Eine Software, die diese Antwort erhält, kann sie analysieren und so umfangreiche Informationen über einen Client abrufen (z.B. die letzten Sensorwerte für jeden angeschlossenen Sensor, etc.) oder zahlreiche verschiedene Aktionen auslösen (z.B. Abschalten aller angeschlossenen Aktoren).

5.10 Datenbank

Das ORCHID Toolkit nutzt eine MySQL-Datenbank zur Datenspeicherung. Von Clients übermittelte Sensordaten werden ebenso in der Datenbank gespeichert wie Instruktionen, die eine Drittsoftware für einen Client hinterlegt. Dieser Abschnitt der Arbeit beschäftigt sich mit dem Aufbau der Datenbank.

Die folgende Abbildung 32 zeigt die Struktur der Datenbank in einer Übersicht.

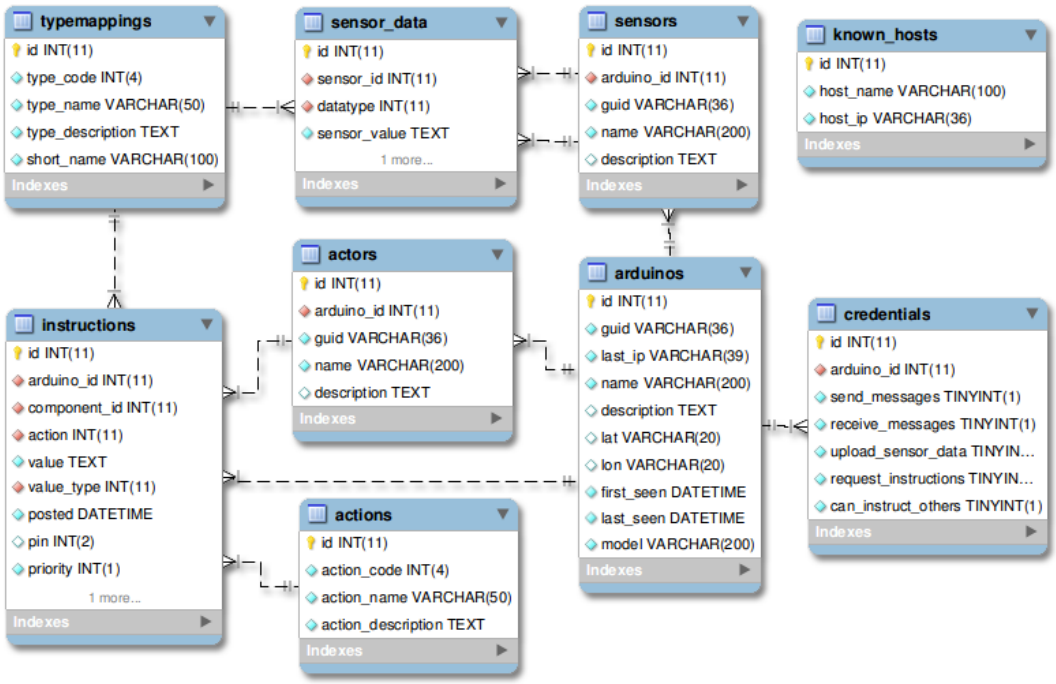

Abbildung 32 - Datenbankstruktur

Im Folgenden werden die einzelnen Tabellen und ihr Verwendungszweck im Detail beschrieben.

Tabelle	actions	
Beschreibung	Speichert mögliche Aktionen, die in einer Instruktion an einen Client zur Ausführung ermittelt werden können und dient entsprechend als Mapping-Tabelle (übertragen wird nur ein numerischer Schlüssel, vgl. 5.8).	
Datenfelder		
id	*INT (11)*	Primärschlüssel
action_code	*INT (4)*	Numerischer Schlüssel, der an Clients übertragen wird.
action_name	*VARCHAR(50)*	Der Name der Aktion.
action_description	*TEXT*	Eine kurze Beschreibung der Aktion.

Tabelle 77 - Datenbanktabelle "actions"

Tabelle	actors	
Beschreibung	Speichert Grundinformationen über Aktoren, die dem ORCHID Server bekannt sind. Jede Zeile in dieser Tabelle steht für einen an einen Client angeschlossenen Aktor.	
Datenfelder		
id	*INT (11)*	Primärschlüssel
arduino_id	*INT (11)*	Fremdschlüssel (*arduinos.id*) Der Client, an den dieser Aktor angeschlossen ist.
guid	*VARCHAR (36)*	Die GUID des Aktors.
name	*VARCHAR (200)*	Der Name des Aktors (frei wählbar).
description	*TEXT*	Die Beschreibung des Aktors (frei wählbar).

Tabelle 78 - Datenbanktabelle "actors"

Tabelle	arduinos	
Beschreibung	Speichert Grundinformationen über Arduino Microcontroller, die einem ORCHID Server bekannt sind. Jede Zeile in dieser Tabelle steht für einen Client.	
Datenfelder		
id	*INT (11)*	Primärschlüssel
guid	*VARCHAR (36)*	Die GUID des Clients.
last_ip	*VARCHAR (39)*	Die letzte IP, mit der sich der Client mit dem Server verbunden hat.
name	*VARCHAR (200)*	Der Name des Clients (frei wählbar).
description	*TEXT*	Die Beschreibung des Clients (frei wählbar).
lat	*VARCHAR (20)*	Der Breitengrad, an dem sich der Client befindet.
lon	*VARCHAR (20)*	Der Längengrad, an dem sich der Client befindet.
first_seen	*DATETIME*	Der Zeitpunkt, zu dem der Client zuerst gesehen wurde.
last_seen	*DATETIME*	Der Zeitpunkt, zu dem der Client zuletzt gesehen wurde.
model	*VARCHAR (200)*	Das Modell des Arduinos, z.B. „Duemilanove“.

Tabelle 79 - Datenbanktabelle "arduinos"

Tabelle	credentials	
Beschreibung	Speichert die Berechtigungen, die ein Client auf dem ORCHID Server hat. Jede Zeile in dieser Tabelle steht für die Berechtigungen eines einzelnen Clients.	
Datenfelder		
id	*INT (11)*	Primärschlüssel
arduino_id	*INT (11)*	Fremdschlüssel (*arduinos.id*) Der Client, dessen Berechtigungen in dieser Zeile abgebildet sind.
send_messages	*TINYINT (1)*	Bestimmt, ob der Client Nachrichten senden darf.
receive_messages	*TINYINT (1)*	Bestimmt, ob der Client Nachrichten empfangen darf.
upload_sensor_data	*TINYINT (1)*	Bestimmt, ob der Client Sensordaten hochladen darf.
request_instructions	*TINYINT (1)*	Bestimmt, ob der Client Instruktionen abfragen darf.
can_instruct_others	*TINYINT (1)*	Bestimmt, ob der Client andere Clients instruieren darf.

Tabelle 80 - Datenbanktabelle "credentials"

Tabelle	instructions	
Beschreibung	Speichert Instruktionen für Clients. Jede Zeile dieser Tabelle steht für eine Instruktion, die von einem Client ausgeführt werden soll oder bereits ausgeführt wurde.	
Datenfelder		
id	*INT (11)*	Primärschlüssel
arduino_id	*INT (11)*	Fremdschlüssel (*arduinos.id*) Der Client, für den die Instruktion bestimmt ist.
component_id	*INT (11)*	Fremdschlüssel (*actors.id*) Der Aktor, für den die Instruktion bestimmt ist.
action	*INT (11)*	Fremdschlüssel (*actions.id*) Die Aktion, die der Aktor, der durch *component_id* identifiziert ist, ausführen soll oder ausgeführt hat.
value	*TEXT*	Der Wert, der an den durch *component_id* identifizierten Aktor übermittelt werden soll oder übermittelt wurde.
value_type	*INT (11)*	Fremdschlüssel (*typemappings.id*) Der Typ des Werts, der an den durch *component_id* identifizierten Aktor übermittelt werden soll oder übermittelt wurde.
posted	*DATETIME*	Der Zeitpunkt, zu dem die Instruktion gespeichert wurde.
pin	*INT (2)*	Der Arduino-I/O-Pin, auf den der Wert *value* geschrieben werden soll (falls kein Pin direkt genutzt werden soll, wird der Wert 999 verwendet).
priority	*INT (1)*	Eine Zahl zwischen 0 und 9, die die Priorität der Instruktion bestimmt (0 ist dabei die höchste Priorität, 9 die niedrigste).
done	*TINYINT (1)*	Gibt an, ob die Instruktion bereits ausgeführt wurde.

Tabelle 81 - Datenbanktabelle "instructions"

Tabelle	known_hosts	
Beschreibung	Beinhaltet alle dem ORCHID Server bekannten Hosts (also IP-Adressen von Clients und Drittsoftware).	
Datenfelder		
id	*INT (11)*	Primärschlüssel
host_name	*VARCHAR (100)*	Der Name des Hosts (frei wählbar).
host_ip	*VARCHAR (36)*	Die dem ORCHID Server bekannte IP-Adresse des Hosts.

Tabelle 82 - Datenbanktabelle "known_hosts"

Tabelle	messages	
Beschreibung	Speichert Nachrichten, die zwischen zwei Clients versendet werden. Jede Zeile dieser Tabelle steht für eine Nachricht zwischen zwei Clients.	
Datenfelder		
id	*INT (11)*	Primärschlüssel
sender_id	*INT (11)*	Fremdschlüssel (`arduinos.id`) Der Client, der die Nachricht gesendet hat.
recipient_id	*INT (11)*	Fremdschlüssel (`arduinos.id`) Der Client, für den die Nachricht bestimmt ist.
msg_text	*TEXT*	Der Text der Nachricht.
posted	*DATETIME*	Der Zeitpunkt, zu dem die Nachricht gespeichert wurde.
received	*TINYINT (1)*	Gibt an, ob die Nachricht empfangen wurde (0 bedeutet, dass sie noch nicht empfangen wurde, 1 bedeutet, dass sie bereits von einem Client abgerufen wurde).
priority	*INT (1)*	Eine Zahl zwischen 0 und 9, die die Priorität der Nachricht bestimmt (0 ist dabei die höchste Priorität, 9 die niedrigste).
Anmerkungen		
Die Tabelle *messages* wird noch nicht verwendet, Nachrichten zwischen Clients sind im Rahmen dieser Arbeit als optionales Kriterium zu verstehen.		

Tabelle 83 - Datenbanktabelle "messages"

Tabelle	sensors	
Beschreibung	Speichert Grundinformationen über Sensoren, die dem ORCHID Server bekannt sind. Jede Zeile in dieser Tabelle steht für einen an einen Client angeschlossenen Sensor.	
Datenfelder		
id	*INT (11)*	Primärschlüssel
arduino_id	*INT (11)*	Fremdschlüssel (`arduinos.id`) Der Client, an den dieser Sensor angeschlossen ist.
guid	*VARCHAR (36)*	Die GUID des Sensors.
name	*VARCHAR (200)*	Der Name des Sensors (frei wählbar).
description	*TEXT*	Die Beschreibung des Sensors (frei wählbar).

Tabelle 84 - Datenbanktabelle "sensors"

Tabelle	sensor_data	
Beschreibung	Speichert gemessene Sensordaten. Jede Zeile in dieser Tabelle steht für einen gemessenen Sensorwert.	
Datenfelder		
id	*INT (11)*	Primärschlüssel
sensor_id	*INT (11)*	Fremdschlüssel (*sensors.id*) Der Datensatz des Sensors, der den Sensorwert gemessen hat.
datatype	*INT (11)*	Fremdschlüssel (*typemappings.id*) Der Datentyp der Daten, die dieser Sensorwert repräsentiert.
sensor_value	*TEXT*	Die eigentlichen Daten, die der Sensor gemessen hat.
measure_time	*DATETIME*	Der Zeitpunkt, zu dem dieser Sensorwert gemessen wurde.

Tabelle 85 - Datenbanktabelle "sensor_data"

Tabelle	typemappings	
Beschreibung	Speichert Datentypen, die bei der Übertragung von Sensordaten und Instruktionen verwendet werden (vgl. 5.8)	
Datenfelder		
id	INT (11)	Primärschlüssel
type_code	INT (4)	Numerischer Schlüssel, der für den Datentyp steht.
type_name	VARCHAR (50)	Der Name des Datentyps.
type_description	TEXT	Eine kurze Beschreibung des Datentyps
short_name	VARCHAR (10)	Kurzer Name des Datentyps.

Tabelle 86 - Datenbanktabelle "typemappings"

6 Realisierung und Tests

Dieser Abschnitt der Arbeit befasst sich mit der Realisierung der in den vorherigen Abschnitten beschriebenen und entwickelten Software, die das ORCHID Toolkit bildet.

Dazu werden zunächst allgemeine Eigenschaften der Implementierung beschrieben und ein Überblick über das Entwicklungssystem gegeben. Im weiteren Verlauf des Abschnitts werden besonders interessante Aspekte der Implementierung hervorgehoben und deren Eigenschaften detailliert dargestellt. Durchgeführte Tests, mit denen die Robustheit und Richtigkeit der entstandenen Software überprüft wurden, werden darüber hinaus ebenso aufgeführt wie nicht gelöste oder nicht lösbare Probleme.

6.1 Styleguide

Der Code, der im Zuge der Entwicklung des ORCHID Toolkits geschrieben wurde, hält sich an zahlreiche Regeln (vgl. 5.5.4 und 5.5.5). Diese umfassen auch Richtlinien für die Gestaltung des Codes und die Namensgebung für Klassen, Methoden und Variablen.

Die grundlegenden Gestaltungsrichtlinien, die während der Entwicklung der Server-Komponente angewendet wurden, werden in diesem Abschnitt beschrieben. Die Gestaltung der Client-Bibliothek weicht von dem hier gezeigten Schema ab.

6.1.1 Einrückungen und Klammerungen

Mit Ausnahme von Zeilen, die aufgrund ihrer Länge ansonsten schlecht lesbar wären, wird nur nach einer öffnenden geschweiften Klammer eingerückt, wobei die Weite der Einrückung vier Leerzeichen beträgt.

Geschweifte Klammern werden stets in einer eigenen Zeile untergebracht; öffnende und schließende geschweifte Klammer sind exakt gleich weit eingerückt. Von dieser Regel ausgenommen sind geklammerte Blöcke, die nur eine einzelne Anweisung enthalten. Optionale geschweifte Klammern werden immer gesetzt.

Das folgende Listing zeigt die wichtigsten Regeln an einem Beispiel.

```
01 public class IndentationDemo
02 {
03     private int _number = 1;
04
05     public int Number
06     {
07         get { return this._number; }
08     }
09
10     public IndentationDemo(int number)
11     {
12         if(number > 0)
13         {
14             this._number = number;
15         }
16         else
17         {
18             this._number = number * -1;
19        }
20     }
21 }
```

Listing 40 - Styleguide zu Einrückungen und Klammern

6.1.2 Sprache und Namenskonventionen

Variablen, Methoden, Klassen und Namespaces werden grundsätzlich mit englischen Namen versehen. Kommentare sind ebenfalls ausschließlich in englischer Sprache verfasst.

Bezeichner werden entweder im *„CamelCase“* oder im *„PascalCase“* geschrieben. *CamelCase* bezeichnet eine Schreibweise, bei der ein Begriff, der eigentlich aus mehreren Wörtern besteht (z.B. *„IndentationDemo“* in Listing 40) in einem Wort zusammengefasst wird und anstatt der Worttrennung mittels Leerzeichen jedes ehemals eigenständige Wort mit einem Großbuchstaben beginnt (aus *„Indentation demo“* wird entsprechend *„IndentationDemo“*). *PascalCase* ist eine Sonderform des CamelCase, bei der auch der Anfangsbuchstabe eines Bezeichners stets ein Großbuchstabe ist.

Die folgende Tabelle 87 zeigt die Namenskonventionen, die im Quellcode verwendet werden.

Element	Schreibweise	Beispiel
Abstrakte Klasse	PascalCase	`AHardwareServer`
Assembly	PascalCase	`Orchid.Core`
Enumeration	PascalCase	`HttpRequestType`
Klasse	PascalCase	`MySqlAdapter`
Klasseneigenschaft	PascalCase	`Port`
Methode	PascalCase	`Start()`
Methodenvariable	CamelCase	`index`
Private Variable	_CamelCase	`_index`
Private Variable (statisch)	_CamelCase	`_log`
Schnittstelle	PascalCase	`IGuid`

Tabelle 87 - Namenskonventionen

6.2 Entwicklungsumgebung

Als Entwicklungssystem kommt ein Einzelplatzrechner mit den folgenden Eigenschaften zum Einsatz:

- **Prozessor:** Intel® Core™ 2 Quad Q6700 @ 2,66 GHz
- **Arbeitsspeicher:** 8 GB
- **Betriebssystem:** Linux Ubuntu 10.10 (*„Maverick Meerkat“*), 64 Bit

Da das ORCHID Toolkit auch unter Microsoft Windows lauffähig sein soll, wird zeitgleich auf einer virtuellen Maschine (realisiert mit Oracle VM VirtualBox 4.1.2) mit den folgenden Eigenschaften entwickelt:

- **Prozessor:** Intel® Core™ 2 Quad Q6700 @ 2,66 GHz (virtuell)
- **Arbeitsspeicher:** 3 GB (virtuell)
- **Betriebssystem:** Microsoft Windows 7 Professional, 32 Bit

Unter Linux wird als Entwicklungsumgebung MonoDevelop 2.4 (Mono 2.6.7), unter Windows Microsoft Visual Studio 2010 Ultimate (.NET Framework 3.5) eingesetzt.

Für die Entwicklung der Client-Bibliothek wird die Arduino IDE (Version 0022) eingesetzt, die für Windows und Linux verfügbar ist.

6.3 Realisierungsdetails

In diesem Abschnitt werden besonders interessante Aspekte der Realisierung des ORCHID Toolkits beschrieben.

6.3.1 Eindeutige Kennungen

Um das ORCHID Toolkit in die Lage zu versetzen, zahlreiche verschiedene Clients und die an diese angeschlossenen Komponenten (Sensoren und Aktoren) verwalten zu können, musste ein Weg gefunden werden, jedes verwaltete Objekt[57] eindeutig zu identifizieren. Eine Verwechslung zweier solcher Objekte hätte im schlimmsten Fall die Beschädigung einer Komponente oder eines Clients zur Folge, weshalb sie in jedem Fall zu vermeiden ist. Zusätzlich gilt, dass die eindeutige Kennzeichnung von Objekten keine spürbar negativen Auswirkungen auf Ausführungsgeschwindigkeit und Stabilität des Toolkits haben darf.

Die verwalteten Objekte werden in einer Datenbank gespeichert, die jedem Objekt automatisch einen Primärschlüssel (eine Ganzzahl) zuweist. Anhand dieser Zahl und der Art des Objekts ist es zwar durchaus möglich, es eindeutig zu identifizieren, jedoch birgt diese Vorgehensweise ein Problem: die Identifizierung von Clients und Komponenten wäre, würde dieser Ansatz verfolgt, abhängig von der Reihenfolge, in der sie in die Datenbank eingefügt werden. Insbesondere wenn ein ORCHID Server mit mehreren Clients kommuniziert, ist dies problematisch, da neue Komponenten einem Clients nur mit vergleichsweise großem Aufwand hinzugefügt werden können.

Um Objekte eindeutig identifizieren zu können, werden deshalb sowohl in der Server-Komponente, als auch in der Client-Bibliothek, *GUIDs (**G**lobally **U**nique **Id**entifiers)* eingesetzt. GUIDs *(dt. etwa: „Weltweit eindeutige Kennungen“)* sind eine Implementierung des UUID-Standards (***U**niversally **U**nique **Id**entifier*) und werden häufig in der verteilten Programmierung eingesetzt, da ihre Eigenschaften sie dafür besonders geeignet erscheinen lassen.

Eine GUID bzw. UUID besteht aus 16 Bytes (128 Bit) und wird in der Regel als Folge von 32 hexadezimalen Ziffern in fünf Gruppen (zu je 8, 4, 4, 4 und 12 Ziffern), getrennt durch Bindestriche dargestellt (insgesamt 36 Zeichen). Eine typische GUID bzw. UUID hat also die folgende Form:

00000000-0000-0000-0000-000000000000

Aufgrund ihrer Länge von 128 Bits existieren 2^{128} (ca. $3{,}403 \times 10^{38}$ oder mehr als 340 *Sextillionen*[58]) verschiedene GUIDs. Diese Zahl ist so groß, dass die Wahrscheinlichkeit, dass zwei GUIDs bzw. UUIDs gleich sind ($P(A) = \frac{1}{2^{128}}$) so gering ist, dass dieses Ereignis als praktisch ausgeschlossen angesehen werden kann.

Das Mono Framework bietet mit `System.Guid` eine stabile und in der Generierung ausreichend schnelle Implementierung des Standards an. Da GUIDs ein zentraler Bestandteil des ORCHID Toolkits sind, wird mit `ggen` (vgl. 9.1) ein Programm zur Generierung solcher Kennungen bereitgestellt. Auf Linux-Systemen, die über das Programm `apt-get` verfügen, kann über den folgenden Befehl zudem ein UUID-Generator installiert werden, dessen Ausgabe als Kennung im Rahmen des Toolkits verwendbar ist.

```
01 apt-get install uuid
```

Listing 41 - Installationsbefehl für uuid (apt-get)

[57] *„Objekt“* bezieht sich hier nicht auf eine Instanz einer Klasse, sondern gilt als Sammelbegriff für Clients, Sensoren und Aktoren.

[58] $2^{128} = 340.282.366.920.938.463.463.374.607.431.768.211.456$

Jeder Client, jeder Aktor und jeder Sensor müssen zur Verwendung mit dem ORCHID Toolkit über eine eindeutige Kennung, also eine gültige GUID, verfügen. Diese Kennung wird bei allen Anfragen an einen ORCHID Server mit übertragen, um das betroffene Objekt eindeutig identifizieren zu können.

6.3.2 Dynamisches Typeloading mit Ninject

Wie bereits in Abschnitt 5.3.3 beschrieben, nutzt die Server-Komponente des ORCHID Toolkits das Entwurfsmuster Dependency Injection. Abschnitt 5.5.2 erläuterte das eingesetzte Dependency Injection Framework Ninject im Detail. In diesem Abschnitt sollen nun Details der Realisierung eines dynamischen Typeloading-Systems unter Verwendung von Ninject betrachtet werden.

6.3.2.1 Definition Typeloading

Unter *Typeloading* (deutsch etwa *„Typen-Laden“*) versteht man das dynamische Laden von Instanzen zur Laufzeit eines Programms. Anstatt direkt im Quellcode ein konkretes Objekt über den Konstruktor seiner Klasse zu erzeugen, wird beim Typeloading stattdessen eine Instanz von einer zentralen Stelle angefordert. Der anfordernden Klasse ist zu diesem Zeitpunkt die konkrete Instanz unbekannt, sie kennt lediglich die (abstrakte) Basisklasse oder eine Schnittstelle.

Für gewöhnlich existiert zu diesem Zweck eine zentrale Verwaltungsklasse, die Methoden bereitstellt, die eine Instanz einer konkreten Klasse auf Wunsch erzeugen und zurückgeben. Dieses Verfahren kann als statisches Typeloading bezeichnet werden, da hier die konkreten Klassen im Code fest kodiert sind und ein Austausch nur durch Veränderung der entsprechenden Codestellen und eine Neukompilierung des Programmcodes möglich ist.

```
01 // Klassische Objekterzeugung
02 SomeType instance = new SomeType();
03
04 // Objekterzeugung mit Typeloading
05 AType instance = TypeDispatcher.CreateInstance();
```

Listing 42 - Beispiel für einfaches Typeloading

Listing 42 zeigt ein Beispiel für einfaches (statisches) Typeloading im direkten Vergleich zur klassischen Objekterzeugung über den Konstruktor der konkreten Klasse `SomeType`. Schon aus diesem einfachen Beispiel wird ersichtlich, dass Typeloading die Kopplung zwischen Klassen stark verringern kann: die Stelle im Code, an der die Objekterzeugung stattfindet, ist unabhängig von der Definition der konkreten Klasse – im Gegenteil zur klassischen Objekterzeugung, bei der die konkrete Klasse bekannt sein muss, da ansonsten der Konstruktor nicht aufgerufen werden kann. Die Klasse `TypeDispatcher` ist im obigen Beispiel dafür zuständig, eine Instanz einer konkreten Implementierung von `AType` zu erzeugen und zurückzugeben.

Dynamisches Typeloading, wie es in der Server-Komponente des ORCHID Toolkits zum Einsatz kommt, geht einen Schritt weiter. Hier werden die konkreten Implementierungen der Klassen, von denen Instanzen über das Typeloading-System erzeugt werden sollen, nicht fest kodiert, sondern deren Typ zur Laufzeit dynamisch bestimmt und erst danach eine Instanz erzeugt.

6.3.2.2 Dynamisches Typeloading in der Server-Komponente

Durch die Verwendung von Ninject als Dependency Injection Framework ist es möglich, das Abstract Factory Pattern so anzupassen, dass es in ein dynamisches Typeloading-System überführt werden kann. Die Funktionsweise dieser Anpassung soll im Folgenden erläutert werden.

Wie in Abschnitt 5.5.2 beschrieben, wird dem Ninject Dependency Injection Framework über ein Modul mitgeteilt, welche abstrakte Basisklasse bzw. welche Schnittstelle an welche konkrete Implementierung gebunden werden soll. Diese Bindung wird in der Server-Komponente durch die Klasse `ServerModule` übernommen, die von der Klasse `NinjectModule` (aus dem Ninject Framework) abgeleitet ist und die notwendige `Load()`-Methode implementiert. `NinjectModule` geht aber noch einen Schritt weiter, als es bei einem regulären Ninject-Modul der Fall ist. Anstatt lediglich eine Liste von Typ-Bindungen (*Typebindings*) abzuarbeiten, enthält diese Modul-Klasse zusätzlich Methoden, die in der Lage sind, mittels *Reflection* einen *Type* anhand dessen Namen und dem Namen der ihn enthaltenen Assembly zu finden.

In einem Ninject-Modul dient die `Load()`-Methode dazu, dem Framework die vorgesehenen Bindungen bekannt zu machen. Im Fall von `NinjectModule` geschieht dies ebenfalls, jedoch werden die Namen der zu bindenden Types zuvor aus der globalen Konfigurationsdatei der Server-Komponente ausgelesen und versucht, diese Types zu finden. Darüber hinaus existieren (optionale) Möglichkeiten, die Stabilität des Moduls zu erhöhen: einerseits wird immer ein Standard-Binding angegeben; andererseits wird, falls ein Typ nicht in einer angegebenen Assembly auffindbar ist, auch in der aktuellen Assembly (`Orchid.Core`) gesucht. Diese Möglichkeiten können über die Konfigurationsdatei ein- und ausgeschaltet werden (vgl. 7.5).

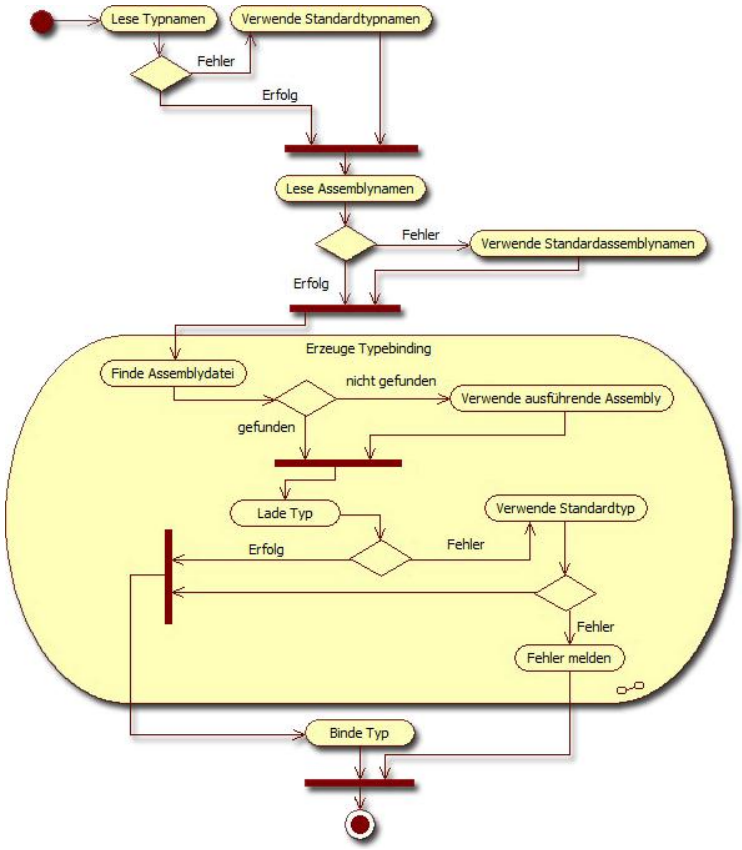

Abbildung 33 - Typeloading

Abbildung 33 stellt in einem Aktivitätsdiagramm dar, wie *Types* über einen Eintrag in der Konfigurationsdatei gefunden und geladen werden. Das Diagramm zeigt den Prozess unter der Prämisse, dass bei nicht auffindbarem *Type* der Standardtyp bzw. bei nicht auffindbarer Assembly die ausführende Assembly verwendet wird.

Kann ein Typ nicht geladen werden, entweder weil er nicht korrekt benannt oder weil die ihn enthaltende Assembly nicht gefunden werden konnte, wird ein Fehler gemeldet und die Ausführung

des Programms entsprechend abgebrochen. So wird bereits beim Laden der zu bindenden *Types* sichergestellt, dass es im weiteren Programmablauf nicht zu Fehlern aufgrund nicht existierender Bindungen kommen kann. Sind auf diese Weise alle Typen erfolgreich geladen worden (und nur dann), werden sie an die abstrakten Basisklassen gebunden, die im restlichen Code der Server-Komponente verwendet werden.

Die hier beschriebene Vorgehensweise zum Typeloading über einen Eintrag in der Konfigurationsdatei erscheint zunächst sehr komplex und aufwändig im Bezug auf die benötigte Rechenzeit. Die Verwendung von Reflection im Code ist stets mit Performanceeinbußen verbunden, da die entsprechenden Methoden rechenintensiv sind. Das Typeloading geschieht im gesamten Programmablauf jedoch nur ein einziges Mal, nämlich während des Startvorgangs des Servers, wenn die erste Instanz eines auf diese Weise geladenen *Types* angefordert wird. Der Performanceverlust kann daher vernachlässigt werden.

6.3.2.3 Dynamisches Typeloading und das Abstract Factory Pattern

Die Eigenschaften des in 5.3.1 beschriebenen Entwurfsmusters *Abstract Factory Pattern* lassen es sehr geeignet für das dynamische Typeloading erscheinen. In der ursprünglichen Form handelt es sich beim Abstract Factory Pattern allerdings um ein statisches Typeloading-System, da die zu verwendenden konkreten Klassen in den konkreten Fabrikklassen und Fabrikmethoden bekannt und entsprechend fest kodiert sein müssen.

Um das Muster in ein dynamisches Typeloading-System zu überführen, muss ein Weg gefunden werden, die konkreten Fabrikklassen von den konkreten Implementierungen der in den Fabrikmethoden erzeugten Instanzen zu entkoppeln. Ziel ist es also, dass auch die konkreten Fabrikklassen lediglich die abstrakten Basisklassen oder Schnittstellen kennen und die eigentlichen Instanzen an anderer Stelle erzeugt werden.

Im vorherigen Abschnitt wurde beschrieben, wie Typen dynamisch geladen werden können. Ohne weitere Klassen wäre es nun möglich, die konkreten Fabrikklassen über einen Ninject-Kernel (vgl. 5.5.2) konkrete Instanzen der zu verwendenden Klassen erzeugen zu lassen, ohne dass diese die konkreten Klassen kennen müssen. Jede Fabrikklasse müsste zu diesem Zweck aber über eine Instanz der Klasse `StandardKernel` aus dem Ninject Dependency Injection Framework verfügen. Eine solche Instanz muss immer auch mindestens ein Modul kennen, es wäre also notwendig, für jede Fabrik ein solches Modul bereitzustellen und zu initialisieren bzw. jeder Instanz ein solches, bereits initialisiertes Modul zu übergeben. Eine solche Doppelung im Code ist nach Möglichkeit zu vermeiden.

Zu diesem Zweck existiert eine weitere Klasse in der Server-Komponente. Es handelt sich dabei um die Singleton-Klasse `NinjectBase`, die einen Ninject-Kernel in einer Singleton-Umgebung bereitstellt. Dieser Kernel kann von allen Fabrikklassen zur Instanzerzeugung verwendet werden.

Zusammenfassend kann der Prozess des dynamischen Typeloadings wie folgt beschrieben werden: Zu Beginn der Programmausführung werden alle Typen, die dynamisch geladen werden, aus der Konfigurationsdatei ausgelesen, mittels Reflection geladen und dann über ein Ninject-Modul innerhalb eines Ninject-Kernels an ihre abstrakten Basisklassen gebunden. Wird an einer beliebigen Stelle im Programmcode eine Instanz einer solchen Klasse angefordert, geschieht dies über eine entsprechende Fabrik. Jede konkrete Fabrik der Server-Komponente greift zur Erzeugung der Instanzen auf einen Ninject-Kernel, der über die Singleton-Klasse `NinjectBase` bereitgestellt wird, zu.

6.3.2.4 Vorteile

Der Hauptvorteil des in den vorherigen Abschnitten beschriebenen dynamischen Typeloading-Verfahrens ist die vollständige Entkopplung der konkreten Klassen voneinander. Wird das reguläre Abstract Factory Pattern eingesetzt, muss ein Entwickler, der eine konkrete Implementierung einer Klasse gegen eine andere austauschen will, in den meisten Fällen auch eine entsprechende konkrete Fabrikklasse implementieren. Diese Anforderung wird durch das dynamische Typeloading mit Ninject hinfällig. Der Austausch einer konkreten Klasse gegen eine andere kann also durch einen einfachen Eintrag in einer Konfigurationsdatei durchgeführt werden, ohne den Quellcode der Server-Komponente verändern zu müssen.

Ein weiterer Vorteil ergibt sich direkt aus dem ersten: die Server-Komponente kann von Entwicklern jederzeit sehr komfortabel erweitert werden, ohne dass diese die ursprüngliche Implementierung modifizieren müssen. Diese Eigenschaft eignet sich insbesondere zum Testen eines Gesamtprojekts, welches das ORCHID Toolkit einsetzt. Ein solches Projekt besteht immer aus mindestens drei Komponenten: einem ORCHID Server, einem ORCHID Client und einer Anwendung, in die die an den Client angeschlossene Hardware integriert werden soll. Wird im Rahmen eines solchen Projekts eine eigene Klasse, z.B. ein eigener Datenbankadapter, entwickelt, kann diese neue Klasse sehr leicht in die bestehende Server-Komponente eingebunden werden. Entsteht während der Entwicklung ein Problem, kann testweise wieder auf die ursprüngliche Klasse gewechselt werden, ohne Änderungen am Code vornehmen zu müssen.

6.3.2.5 Nachteile

Der wohl gravierendste Nachteil des beschriebenen Verfahrens ist, dass durch die Verwendung zahlreicher verschiedener Klassen, die für das dynamische Typeloading notwendig sind, die Lesbarkeit des Quellcodes eingeschränkt wird. Dieser Nachteil wird aber durch die hohe Dynamik und die strenge Entkopplung, die das Verfahren mit sich bringt, relativiert.

Die durch die Verwendung von Reflection auftretenden Performanceeinbußen sind durch ihr frühes Auftreten im Programmablauf und ihre Einmaligkeit vernachlässigbar gering und werden an dieser Stelle nur aus Gründen der Vollständigkeit genannt.

Darüber hinaus ist das Laden eines Typs über dessen Namen immer mit dem Risiko verbunden, dass es zu Fehlern kommen kann. Solche Fehler können z.B. auftreten, wenn der Typname nicht korrekt oder nicht vollständig angegeben wird. Geschieht dies zur Laufzeit eines Programms, kann es durch nicht initialisierte Objektreferenzen zu Fehlern Abstürzen kommen. Da das Typeloading-System der Server-Komponente aber über Sicherheitsmaßnahmen und Integritätsprüfungen verfügt, können solche Fehler oder gar Abstürze der Software ausgeschlossen werden.

6.3.3 Sicherheitsaspekte

Da das ORCHID Toolkit für die Entwicklung von Prototypen ausgelegt ist und Projekte, die auf dem Toolkit basierend entwickelt werden, nicht in Produktivumgebungen eingesetzt werden sollen, kann auf Sicherheitsfunktionen weitgehend verzichtet werden.

Da aber ein ORCHID Server unter Umständen von einem Entwickler nicht im lokalen Netzwerk, sondern im Internet betrieben wird, oder auf andere Weise über das Internet erreichbar ist, ist es sinnvoll, ein Mindestmaß an Sicherheitsfunktionen zu implementieren, die es einem potentiellen Angreifer schwieriger machen, Daten von einem ORCHID Server abzurufen oder Daten auf einen ORCHID Server zu laden.

6.3.3.1 Credentials

Die Server-Komponente ist in der Lage, jedem Client, der ihr bekannt ist, Berechtigungen (*„Credentials“*) zuzuweisen (vgl. 5.10). Möchte ein Client eine bestimmte Aktion durchführen, z.B. Sensordaten auf den Server laden, wird zunächst geprüft, ob dieser Client über die entsprechende Berechtigung verfügt. Ist dies nicht der Fall, wird die Anfrage als nicht autorisierter Zugriff (HTTP-Code 401, *„Unauthorized“*) abgewiesen. Ansonsten wird die geforderte Aktion durchgeführt.

6.3.3.2 Developer Keys

Das Prinzip der Developer Keys *(dt. etwa: „Entwickler-Schlüssel“)* entspricht im Wesentlichen der Verwendung eines Passworts. Ein Entwickler, der ein Projekt auf Basis des ORCHID Toolkits entwickelt, kann sich einen Developer Key generieren lassen (ein entsprechendes Programm liegt dem Toolkit bei) und diesen in Anfragen an einen ORCHID Server übertragen. Der Server vergleicht den übertragenen Developer Key mit dem ihm bekannten, zulässigen Schlüssel und weist alle Anfragen, die diesen Schlüssel nicht übertragen, mit einer Fehlermeldung zurück (HTTP-Code 401, *„Unauthorized“*).

Ein Developer Key kann – theoretisch – jede beliebige Zeichenfolge sein. Die Verwendung des mitgelieferten Programms `dkgen` wird jedoch empfohlen, da dieses Programm einzigartige (im Sinne von *„unique“*) Schlüssel generiert, die lang genug sind, um mit ausreichend hoher Wahrscheinlichkeit nicht erraten werden zu können.

```
steps@zombie:~/Masterarbeit/svn/trunk/dkgen/bin/Debug$ ./dkgen.exe
543cb80b14e00c16c3d19e0c8ee82aaf448ba47fbaa6b9b011c5069fe62e4277f12a0ed9
```

Abbildung 34 - Ausgabe von dkgen

Die obige Abbildung 34 zeigt beispielhaft die Ausgabe von `dkgen`. Ein auf diese Weise erzeugter Developer Key besteht aus 72 hexadezimalen Zeichen. Es ist nahezu unmöglich, einen solchen Schlüssel zu erraten, um so Daten von einem ORCHID Server zu beziehen oder Daten auf einen ORCHID Server zu laden; ein mit `dkgen` erzeugter Developer Key bietet also ein gewisses Mindestmaß an Sicherheit, kann jedoch auch nicht vollständig vor unbefugten Zugriffen schützen (vgl. 6.6.1).

6.3.3.3 Known IPs

Als dritte und letzte Sicherheitsmaßnahme kann ein ORCHID Server so konfiguriert werden, dass ausschließlich Verbindungen von bekannten IP-Adressen *(engl.: „known IP addresses“ oder kurz „known IPs“)* angenommen werden. Jegliche Anfrage, die nicht von einer dem ORCHID Server bekannten IP-Adresse gesendet wurde, wird in diesem Fall mit einer Fehlermeldung (HTTP-Code 401, *„Unauthorized“*) abgelehnt.

Auch die Verwendung dieser Maßnahme kann nicht vollständig vor unbefugten Zugriffen auf einen ORCHID Server schützen, da IP-Adressen vorgetäuscht werden können (Details werden in 6.6.1 beschrieben).

6.3.3.4 Fazit

Die drei in diesem Abschnitt geschilderten Sicherheitsmaßnahmen Credentials, Developer Keys und Known IPs ergeben ein Mindestmaß an Sicherheit, können jedoch einen unbefugten Zugriff auf einen ORCHID Server nicht vollständig ausschließen.

6.4 Testfälle und Auswertung

Im Folgenden werden die Tests beschrieben, die durchgeführt wurden, um die Richtigkeit und Fehlerfreiheit der entwickelten Software gewährleisten zu können.

Getestet wurde einerseits, ob die Bestandteile des Toolkits fehlerfrei und richtig arbeiten und berechnete Ergebnisse den erwarteten entsprechen (Unit-Tests), andererseits, ob die Bestandteile auch unter Belastung noch wie vorgegeben arbeiten (Lasttests).

Da die in den folgenden Abschnitten beschriebenen Tests sehr umfangreich sind, handelt es sich hierbei nicht um eine vollständige Liste, sondern um eine zusammenfassende Beschreibung. Die durchgeführten Tests befinden sich vollständig auf dem dieser Arbeit beiliegenden Datenträger (im Verzeichnis `Orchid/Tests`).

6.4.1 Unit-Tests (Server-Komponente)

Die durchgeführten Unit-Tests stellen sicher, dass alle Bestandteile der Server-Komponente fehlerfrei und richtig arbeiten; sie wurden unter Verwendung des *NUnit*-Frameworks[59] implementiert.

6.4.1.1 Server-Tests

Unter dem Begriff „Server-Tests“ werden alle Unit-Tests zusammengefasst, die prüfen, ob die Server-Komponente eingehende Anfragen korrekt bearbeitet und beantwortet. Dazu werden für alle Arten von Requests, die die Server-Komponente verarbeiten kann (vgl. 5.8) gültige und ungültige Anfragen erzeugt, an eine laufende Server-Instanz übermittelt und die entsprechenden Antworten ausgewertet.

Die Server-Komponente verhält sich dem definierten Protokoll entsprechend. Fehlerhafte oder unvollständige Anfragen führen nicht zu unvorhergesehenen Konsequenzen oder falsch-positiven Ergebnissen[60]. Programmabstürze treten nicht auf.

6.4.1.2 Parser-Tests

Die zentral an der Bearbeitung eingehender Anfragen beteiligte Parser-Klasse muss eingehende Datenströme fehlerfrei auslesen und in Request-Objekte überführen. Um dies zu testen werden laut Protokoll (vgl. 5.8) gültige und ungültige, sowie zufällig generierte Anfragen an die Klasse übermittelt und die von dieser erzeugten Request-Objekte auf deren Gültigkeit hin überprüft.

Die Parser-Klasse arbeitet fehlerfrei. Auch unvollständige oder fehlerhafte Anfragen werden erwartungsgemäß in Request-Objekte überführt.

6.4.1.3 Konfigurationstests

Die Funktionsfähigkeit der Server-Komponente hängt direkt mit der Konfiguration zusammen. Diese muss entsprechend fehlerfrei arbeiten. Für diesen Test werden alle in der Standard-Konfigurationsdatei vorhandenen Werte ausgelesen und überprüft.

Das Konfigurationssystem arbeitet fehlerfrei, fehlende Einstellungen werden durch voreingestellte Werte ersetzt.

6.4.1.4 Typeloading-Tests

Die Server-Komponente erzeugt Instanzen über das integrierte dynamische Typeloading-System. Dieses muss in der Lage sein, Types aus Assemblies zu laden und fehlerhafte oder fehlende Angaben ggf. durch Standardwerte zu ersetzen. Um das Typeloading zu testen, werden Instanzen aller Klassen, die in der Server-Komponente dynamisch geladen werden, erzeugt und überprüft.

---

[59] http://www.nunit.org/

[60] Ergebnisse, die aufgrund eines fehlerhaften Requests nicht hätten ermittelt werden dürfen.

Das Typeloading-System arbeitet fehlerfrei, unzureichende oder fehlerhafte Angaben werden zuverlässig durch voreingestellte Standardwerte ersetzt. Der verwendete Mechanismus hat keine spürbaren Auswirkungen auf die Ausführungsgeschwindigkeit der Server-Komponente.

6.4.1.5 Fabrik-Tests

Die Fabrikklassen müssen Objekte zuverlässig und vorhersagbar erzeugen. Unerwartete `null`-Werte dürfen nicht auftreten. Zwei erzeugte Objekte des gleichen Typs müssen die gleichen Werte beinhalten. Für diesen Test werden alle über Fabriken zu erzeugenden Klassen über die entsprechende Fabrik instanziiert und die Instanzen auf ihre Gültigkeit überprüft.

Alle Fabrikklassen arbeiten fehlerfrei und erzeugen Objekte, die den gestellten Erwartungen entsprechen.

6.4.2 Unit-Tests (Client-Bibliothek)

Die für die Client-Bibliothek implementierten Unit-Tests liegen in Form eines Test-Sketchs vor, der alle Methoden der Bibliothek aufruft und auf die Richtigkeit der zurückgegebenen Werte prüft. Dieser Sketch unterscheidet, ob für einen laufenden ORCHID Server getestet werden soll, oder ob nicht herstellbare Verbindungen der erwartete Fall sind.

Alle Methoden der Client-Bibliothek arbeiten wie erwartet und geben stets die richtigen Werte und Fehlercodes zurück.

6.4.3 Lasttests

Lasttests werden ausschließlich für die Server-Komponente ausgeführt. ORCHID Clients werden per Definition nicht stark belastet, da sie zu jedem Zeitpunkt mit maximal einem Server kommunizieren und die Kommunikation stets von ihnen initialisiert wird.

Die durchgeführten Lasttests prüfen, ob die Server-Komponente auch unter stärkerer Belastung noch voll funktionsfähig ist und die in 4.6 beschriebenen Leistungsvorgaben erfüllen kann. Zu diesem Zweck wurde das Testprogramm *„`loadtest.sh`“* (ein Shellscript für die Bash[61]) entwickelt, dass auf einem ORCHID Server eine große Last erzeugen kann (entspricht dem nebenläufigen Zugriff mehrerer Clients).

Auch unter großer Last (bis zu 20 Anfragen pro Sekunde) arbeitet die Server-Komponente fehlerfrei.

6.5 Probleme während der Entwicklung

Aufgrund der guten Dokumentation sowohl des Mono Frameworks, als auch der Arduino Microcontroller-Plattform, traten während der Entwicklung kaum Probleme auf. Ein besonders schwierig zu lösendes Problem sei aber an dieser Stelle genannt, da dessen Lösung wichtig für potentielle Weiterentwicklungen des Toolkits ist.

Die Client-Bibliothek des Toolkits wurde in C++ entwickelt, da dies die Programmiersprache ist, mit der Bibliotheken für den Arduino Microcontroller implementiert werden. Ein Debugger oder ähnliche Werkzeuge, die bei der Fehlersuche hilfreich hätten sein können, stehen für die Programmierung von Microcontrollern nicht zur Verfügung. Speicher muss vom Programmierer selbst verwaltet werden, dies betrifft insbesondere die Zuweisung und die Freigabe von Speicherbereichen im Hauptspeicher.

Die erste Version der Client-Bibliothek funktionierte zunächst fehlerfrei, war aber nicht in der Lage, die Antwort eines Servers zu lesen und zu interpretieren; diese Funktion wurde nachträglich

[61] http://tiswww.case.edu/php/chet/bash/bashtop.html

implementiert. Erste Tests zeigten, dass die Kommunikation zwischen Client und Server reibungslos ablief.

Nach der Implementierung der Funktion, Instruktionen von einem ORCHID Server abzurufen, verhielt sich der ausführende Arduino jedoch nicht mehr wie vorgesehen. Die Hauptmethode des Sketchs, der zum Testen geschrieben wurde, konnte nicht mehr bis zum Ende ausgeführt und dann wiederholt werden. Stattdessen brach die `loop()`-Methode immer wieder ab und die `setup()`-Methode, die per Definition nur einmal zu Beginn der Ausführung aufgerufen wird (und auch nicht manuell aufgerufen werden kann) wurde ausgeführt. Statt also lediglich die `loop()`-Methode zu wiederholen, führte der Arduino beide Methoden in jedem Durchlauf aus.

Eine ausgiebige Fehleranalyse identifizierte das Auslesen und Verwerten der Server-Antwort als Problem. Der Verdacht lag nah, dass die Allokation von Speicherbereichen für Datenpuffer das vom Standard abweichende Verhalten verursachen könnte. Wahrscheinlich würde der belegte Speicher nicht ordnungsgemäß freigegeben oder läge in einem Speicherbereich, der nicht beschrieben werden dürfte. Eine testweise Ausgabe belegte diesen Verdacht, da anstatt der zu verwertenden Antwort des Servers lediglich Sonderzeichen ausgegeben wurden (dies ist häufig der Fall, wenn ein Speicherbereich, der nicht dafür vorgesehen ist, als Zeichen interpretiert wird). Den Speicher nach der Verwendung wieder freizugeben löste das Problem jedoch nicht.

Die Zeichenkettenverarbeitung in C++ unterscheidet sich erheblich von der in moderneren Programmiersprachen. Insbesondere muss ein Entwickler darauf achten, dass eine Zeichenkette, die er selbst aus einzelnen Zeichen zusammensetzt (wie es in der Client-Bibliothek der Fall ist), stets mit einem Null-Byte endet. Die Zeichenkettenfunktionen von C++ lesen eine Zeichenkette so lange, bis ein solches Null-Byte erreicht wird – fehlt dieses Byte, lesen die Funktionen so lange weiter, bis ein Null-Byte gefunden wird oder der entsprechende Puffer voll ist. Dieses Verhalten kann auch dazu führen, dass Programme abstürzen. Das Hinzufügen eines Null-Bytes am Ende des verwendeten Puffers konnte das Problem jedoch auch nicht beheben.

Letztendlich war die Ursache des Problems – und auch dessen Lösung – sehr simpel. Die Methode, die eine Server-Antwort auslesen und verarbeiten sollte, reservierte 1024 Bytes für den Puffer, in den die Antwort zunächst eingelesen werden sollte. Im Normalfall ist die Reservierung dieser Menge an Speicher auch problemlos möglich; ein Arduino Microcontroller (Modell „Duemilanove“) verfügt jedoch über lediglich 2048 Bytes an Arbeitsspeicher – die Methode reservierte also die Hälfte des überhaupt verfügbaren Speichers. Andere Programmteile und Daten belegen weitere Bereiche, so dass letztendlich mehr Speicher reserviert werden sollte, als überhaupt verfügbar war. Entsprechend stürzte der Sketch an dieser Stelle ab und der Arduino setzte sich zurück – was zum Abbruch der `loop()`-Methode und zum Aufruf von `setup()` führte.

Die Speicherreservierung für den Datenpuffer wurde verkleinert und das Problem damit behoben. In der aktuellen Version reserviert die Methode 128 Bytes.

6.6 Offene Probleme

Im Folgenden werden Probleme beschrieben, die sich während der Entwicklung des Toolkits ergeben haben und im Rahmen dieser Arbeit nicht gelöst werden konnten. Kritische Probleme, die einen Misserfolg der Arbeit zur Folge hätten, sind nicht aufgetreten.

6.6.1 Sicherheitsbedenken

Sicherheitsfunktionen wie die Verschlüsselung des Datenverkehrs sind nicht Bestandteil dieser Arbeit. Die Daten, die zwischen einem ORCHID Client und einem ORCHID Server ausgetauscht werden, werden also im Klartext übermittelt.

Jeder, der sich im gleichen Netzwerk befindet, in dem Client und Server miteinander kommunizieren, kann die ausgetauschten Daten mitlesen (darunter auch den Developer Key). Die folgende Abbildung zeigt ein Bildschirmfoto der Netzwerk-Analysesoftware *„Wireshark“*[62].

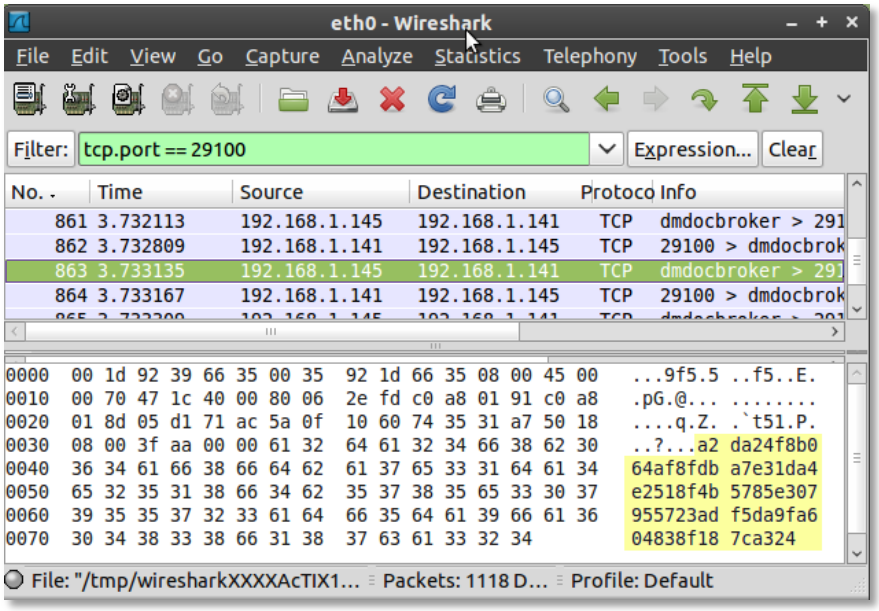

Abbildung 35 - Bildschirmfoto "Wireshark"

Klar zu erkennen ist der (in der Abbildung gelb hervorgehobene) im Klartext übermittelte Developer Key. Auch alle anderen übermittelten Daten können auf diese Weise aus dem Netzwerk mitgelesen werden.

Kennt ein Dritter den Developer Key und die benötigten GUIDs von Client und Sensor/Aktor, kann er Anfragen an den ORCHID Server fälschen. Ist das Protokoll bekannt, das vom ORCHID Toolkit eingesetzt wird, können so z.B. Instruktionen für einen ORCHID Client auf einem ORCHID Server hinterlegt werden, die der ursprüngliche Entwickler nicht vorgesehen hat.

Da ein ORCHID Client in der Lage ist, Hardware zu steuern, kann eine bösartige Instruktion auf dem Server zu erheblichem Schaden führen, wenn z.B. ein Küchengerät (etwa ein Toaster) mit dem Toolkit angesteuert werden kann. Solche und ähnliche Geräte können potentiell einen Brand auslösen, wenn sie nicht ordnungsgemäß oder ohne Aufsicht betrieben werden. Ein anderes Szenario, in dem eine eingeschleuste Instruktion zu erheblichem Schaden führen kann, ist die Steuerung eines elektronischen Schlosses, z.B. an einer Wohnungstür. Ein Angreifer wäre theoretisch in der Lage, dieses Schloss zu öffnen.

Die vorliegende Implementierung des Toolkits verfügt über verschiedene Mechanismen, fehlerhafte oder bösartige Anfragen von nicht autorisierten Clients zu erkennen und abzuweisen; diese Mechanismen können aber umgangen werden, wenn ein Angreifer wie in diesem Abschnitt beschrieben den Developer Key eines Entwicklers erfährt. Auch die Sicherheitsmaßnahme, nur Anfragen von bekannten IP-Adressen entgegenzunehmen und zu verarbeiten, kann umgangen werden, indem ein Angreifer *IP-Spoofing (dt. etwa: „IP-Verschleierung“* oder *„IP-Manipulation“)* einsetzt. Dabei werden Datenpakete so manipuliert, dass sie als Absender-IP-Adresse nicht mehr die wirkliche Adresse des Absenders, sondern eine beliebige von diesem gewählte IP-Adresse enthalten.

[62] http://www.wireshark.org/

Zusammenfassend kann festgehalten werden, dass das ORCHID Toolkit um eine Möglichkeit zur Verschlüsselung der zwischen Client und Server versendeten Daten erweitert werden muss, um für sicherheitskritische Einsatzbereiche geeignet zu sein.

6.6.2 Stabilitätsbedenken

Durch den für die Entwicklung einer Bibliothek für die Arduino-Plattform notwendigen Einsatz von C++ als Programmiersprache ergeben sich zum Teil Probleme, die unter bestimmten Umständen zu Stabilitätsproblemen der Client-Bibliothek führen können. In Abschnitt 6.5 wurde ein solches Problem bereits beschrieben (dieses spezielle Problem ist aber bereits gelöst).

Weitere Probleme dieser Art sind zu befürchten, da besonders fehlertoleranter Code einen Großteil des zur Verfügung stehenden Speichers verbrauchen würde (vgl. 6.6.4).

Während der Entwicklung der Client-Bibliothek wurde stets darauf geachtet, typische Fehler (Zugriffe auf Array-Indizes außerhalb des definierten Bereichs, Puffer-Überläufe, etc.) zu vermeiden, jedoch können Stabilitätsprobleme dadurch nicht vollständig ausgeschlossen werden.

6.6.3 Stabilitätsbedenken (Ethernet-Bibliothek)

Die mit der Arduino-IDE ausgelieferte Ethernetbibliothek kann nicht als ausgereifte Software bezeichnet werden. In ihr kann vielmehr der Versuch erkannt werden, eine möglichst schnell verfügbare Bibliothek zu entwickeln, um mit einem Arduino Microcontroller über das Netzwerk kommunizieren zu können.

Die Bibliothek ist nur rudimentär kommentiert und bietet keinerlei Fehlerbehandlung an, weshalb von Zeit zu Zeit Probleme auftreten können. Diese äußern sich meist darin, dass ein Client sich nicht zu einem Server verbinden kann, ohne dass dafür ein Grund erkennbar ist. Dieses Verhalten ist zum Teil vollständig unvorhersehbar: es kann vorkommen, dass eine oder zwei Verbindungen erfolgreich zustande kommen, während der dritten aber ein Fehler auftritt und danach keine weitere Verbindung mehr möglich ist. Nur ein Rücksetzen des Arduinos behebt diesen Fehler.

Es wurde während der Implementierung der Client-Bibliothek versucht, die Ethernet-Bibliothek so fehlertolerant wie möglich zu nutzen und auftretende Fehler ggf. zu umgehen. So konnte eine durchaus stabile Bibliothek geschaffen werden – Fehler durch die Ethernet-Bibliothek können aber weiterhin nicht ausgeschlossen werden.

6.6.4 Speicherprobleme

Wie bereits erwähnt, kann es aufgrund von Speichermangel zu Problemen mit einem Arduino Microcontroller kommen. Während der Entwicklung zeigten sich diese Probleme hauptsächlich durch die in 6.5 bereits diskutierten Fehler in der Ausführung eines Sketchs.

Da aber nicht nur der Arbeitsspeicher eines Arduinos begrenzt ist, sondern auch nur wenig Festspeicher zur Verfügung steht (vgl. 4.3.2.2 und [5]), entstehen zusätzlich weitere Probleme. Insbesondere bei der Verarbeitung von Antworten eines Servers würde mehr belegbarer Speicher dazu führen, dass Fehlerbehandlungsroutinen und im Allgemeinen robusterer Code implementiert werden könnten. Da die Client-Bibliothek des Toolkits jedoch nur einen möglichst kleinen Anteil an der Gesamtgröße eines Sketchs haben soll, musste auf solche Routinen weitgehend verzichtet werden. Entsprechend muss das in 5.8 beschriebene Protokoll strikt eingehalten werden, damit die Kommunikation zwischen Client und Server reibungslos ablaufen kann.

7 Handbuch Server-Komponente

Dieser Abschnitt bildet das Benutzerhandbuch für die Server-Komponente des ORCHID Toolkits. Nachdem zunächst die Voraussetzungen beschrieben werden, die ein System, auf dem die Server-Komponente ausgeführt werden soll, erfüllen muss, wird auf die Installation, Konfiguration und Verwendung der Komponente eingegangen. Im Verlauf dieses Handbuchs wird dazu eine beispielhafte Referenzimplementierung für einen ORCHID Server entwickelt.

Abschließend wird gezeigt, wie die Server-Komponente erweitert bzw. verändert werden kann, indem eine neue Klasse entwickelt wird, die eine bereits vorhandene ersetzt.

Falls nicht anders angegeben, befinden sich alle genannten Dateien und Verzeichnisse auf der DVD, die dieser Arbeit beiliegt (Details zum beiliegenden Datenträger können Abschnitt 17 entnommen werden).

7.1 Systemvoraussetzungen

Zur Ausführung wird ein installiertes Mono Framework (Version 2.6.7 oder kompatibel) benötigt. Installationsdateien für Microsoft Windows, Linux und Apple MacOS befinden sich auf der beigelegten DVD. Alternativ kann die für das entsprechende System benötigte Installationsdatei von der Herstellerseite[63] heruntergeladen werden.

Zweite Voraussetzung ist eine erreichbare Instanz eines MySQL-Servers, d.h. eine Instanz, auf die vom ausführenden System aus zugegriffen werden kann.

Weitere Anforderungen an Hard- und Software des ausführenden Systems können Abschnitt 4.3.1.1 bzw. 4.3.2.1 entnommen werden.

7.2 Installation

Die Server-Komponente erfordert keine gesonderte Installation. Es ist ausreichend, die Binärdateien vom beiliegenden Datenträger zu kopieren und an einem beliebigen Ort im Dateisystem abzulegen. Die ausführbaren Dateien der Server-Komponente und alle weiteren zur Ausführung benötigten Bibliotheken befinden sich im Order `Orchid/Server/bin/lib` auf der DVD.

Es ist ausreichend, den Ordner `Orchid/Server/bin/lib` zu kopieren und im Dateisystem abzulegen. Er enthält die folgenden Dateien:

Datei	Verwendung
`MySql.Data.dll`	Schnittstelle zu MySQL-Datenbanken
`Nini.dll`	Konfigurationsframework
`Ninject.dll`	Dependency Injection Framework
`NLog.dll`	Logging-Framework
`Orchid.Core.dll`	Server-Komponente des ORCHID Toolkits

Tabelle 88 - Binärdateien der Server-Komponente

Die Server-Komponente referenziert alle anderen Assemblies, entsprechend müssen sie für ein Projekt, das die Komponente einbindet, auffindbar sein (es ist ausreichend, die im gleichen Verzeichnis abzulegen).

[63] http://www.go-mono.com/mono-downloads/download.html

7.2.1 Installation der MySQL-Datenbank und des MySQL-Servers

Für den Betrieb der Server-Komponente ist eine vom ausführenden System aus erreichbare Instanz eines MySQL-Servers notwendig. Die Installation eines solchen Servers ist von System zu System verschieden; dieser Abschnitt geht auf die Installation unter Ubuntu Linux 10.10 und Microsoft Windows 7 ein.

7.2.1.1 Installation unter Ubuntu Linux 10.10

Unter Ubuntu Linux stehen zwei Möglichkeiten zur Installation zur Verfügung. Entweder wird ein MySQL-Server über den Paketmanager des Systems oder das XAMPP-Server-Paket[64] installiert. Da das XAMPP-Server-Paket auch einen Apache Webserver beinhaltet, der für die Ausführung der Proof of Concept Software notwendig ist (vgl. Abschnitt 10), empfiehlt es sich, dieses Paket zu installieren. Die Installation wird im Folgenden beschrieben.

Zunächst muss das für das jeweilige System passende Paket heruntergeladen[65] und an einem beliebigen Ort im Dateisystem (z.B. `/home/<benutzername>/Downloads`) abgelegt werden. Dieser Speicherort muss nun im Terminal (der Kommandozeile) geöffnet werden.

Innerhalb des Verzeichnisses kann nun der Befehl zum Entpacken des heruntergeladenen Archivs ausgeführt werden, den die folgende Abbildung zeigt:

```
orchid@orchidvm ~/Downloads $ sudo tar xvfz xampp-linux-1.7.4.tar.gz -C /opt
```

Abbildung 36 - Installation von XAMPP for Linux

Dieser Befehl extrahiert die Dateien in `xampp-linux-1.7.4.tar.gz` in das Verzeichnis `/opt` (aus diesem Grund müssen Administratorrechte vorhanden sein, weshalb der Befehl mittels `sudo` ausgeführt werden muss). Nachdem die Dateien extrahiert wurden, ist XAMPP für Linux installiert. Die Datei `xampp-linux-1.7.4.tar.gz` kann gelöscht werden.

Die Dateien von XAMPP befinden sich im Verzeichnis `/opt/lampp`.

7.2.1.2 Installation unter Microsoft Windows 7

Die Installation von XAMPP unter Microsoft Windows 7 wird mittels eines Installationsprogramms durchgeführt. Es genügt, dieses Programm herunterzuladen, auszuführen und den Anweisungen auf dem Bildschirm zu folgen.

Nach der Installation befinden sich die Dateien von XAMPP im Verzeichnis `C:\xampp` (falls bei der Installation kein anderes Verzeichnis angegeben wurde).

7.2.1.3 Import der Datenbankstruktur

Die dieser Arbeit beiliegende DVD enthält ein Verzeichnis namens `Orchid/Database`. Dieses Verzeichnis enthält eine Datei mit dem Namen „`orchid.sql`“. Bei dieser Datei handelt es sich um die Datenbankstruktur (und die notwendigen Mindestdaten), die zum Betrieb eines ORCHID Servers notwendig sind (vgl. 5.10).

Der Import der Datenbank in die auf dem eigenen System installierte MySQL-Instanz wird mit dem Kommandozeilenprogramm `/opt/lampp/bin/mysql` (`C:\xampp\mysql\bin\mysql.exe` unter Windows) durchgeführt (der Pfad zum Programm kann je nach den bei der Installation von XAMPP gemachten Angaben variieren). Die folgenden Abbildungen zeigen den Prozess des Imports.

---

[64] http://www.apachefriends.org/
[65] http://www.apachefriends.org/de/xampp.html

```
orchid@orchidvm ~/orchid_server $ /opt/lampp/bin/mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 742
Server version: 5.5.8 Source distribution

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE DATABASE orchid;
Query OK, 1 row affected (0.00 sec)

mysql> quit
Bye
orchid@orchidvm ~/orchid_server $
```

Abbildung 37 - Erzeugung der MySQL-Datenbank

Die obige Abbildung 37 zeigt, wie mittels des Programms `mysql` eine Datenbank namens „`orchid`“ auf dem lokalen System erzeugt wird. Über das gleiche Programm kann die Datei „`orchid.sql`“ von der DVD in die lokale MySQL-Instanz importiert werden:

```
orchid@orchidvm ~/orchid_server $ /opt/lampp/bin/mysql -uroot -p orchid < orchid
.sql
Enter password:
orchid@orchidvm ~/orchid_server $
```

Abbildung 38 - Import der Datenbankstruktur

Nachdem dieser Befehl ausgeführt wurde, kann die lokale MySQL-Instanz als Datenbank für die Server-Komponente verwendet werden.

7.3 Referenzierung

Um die Server-Komponente in einem Softwareprojekt nutzen zu können, muss die Assemblydatei `Orchid.Core.dll` durch das Projekt referenziert werden. Wie bereits erläutert, müssen auch die übrigen Assemblydateien aus dem `bin`-Verzeichnis der Server-Komponente für das referenzierende Projekt auffindbar sein.

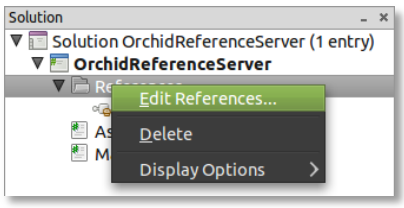

Abbildung 39 - Referenzierung der Server-Komponente

Die obige Abbildung zeigt, wie einem Projekt in der Entwicklungsumgebung MonoDevelop eine Referenz hinzugefügt werden kann. In anderen Entwicklungsumgebungen ist der Ablauf analog.

Im folgenden Dialog muss zu der Assemblydatei `Orchid.Core.dll` navigiert und diese als neue Referenz für das Projekt ausgewählt werden, wie in Abbildung 40 dargestellt ist.

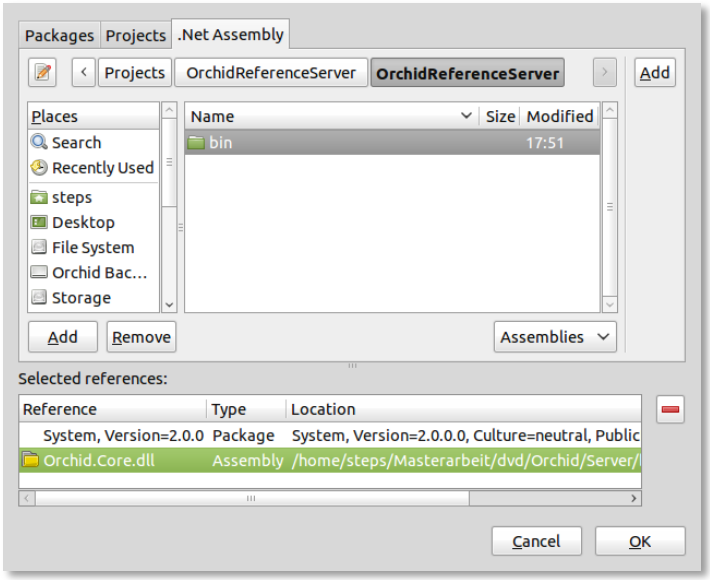

Abbildung 40 - Referenzierungsdialog von MonoDevelop

Nachdem der Dialog bestätigt wurde, ist `Orchid.Core.dll` als Referenz hinzugefügt worden und kann entsprechend verwendet werden.

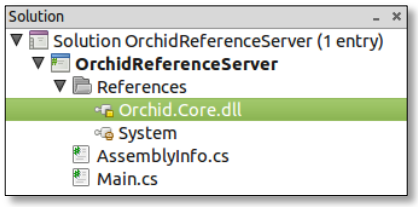

Abbildung 41 - Referenzierte Server-Komponente

Die obige Abbildung zeigt die Referenzen des Projekts, nachdem `Orchid.Core.dll` erfolgreich hinzugefügt wurde.

7.4 Minimalprojekt

Dieser Abschnitt beschreibt die Schritte, die mindestens notwendig sind, um eine Software zu entwickeln, die als ORCHID Server eingesetzt werden kann.

Wie 5.6 erläutert, bildet die Klasse `Bloom` den zentralen Einstiegspunkt in die Server-Komponente. Entsprechend muss eine Instanz dieser Klasse erzeugt werden, um einen funktionierenden ORCHID Server zu entwickeln. Das folgende Listing 43 zeigt, wie eine Instanz der Klasse erzeugt wird. Das in diesem Listing gezeigte Programm ist ein vollständiger ORCHID Server, der Anfragen von Clients und Drittsoftware entgegennehmen, verarbeiten und beantworten kann. Dieses Beispiel zeigt, wie einfach es ist, die Server-Komponente in eigene Projekte einzubinden.

```
01 using System;
02 using Orchid.Core;
03 using Orchid.Core.Configuration;
04 using Orchid.Core.Daemons;
05 using Orchid.Core.Servers;
06
07 namespace OrchidReferenceServer
08 {
09     public static class MainClass
10     {
11         public static void Main (string[] args)
12         {
13             string cfgFile = "Settings/Orchid.ini";
14
15             // Initialize configuration system
16             Config.Instance.Initialize(new FileInfo(cfgFile));
17
18             // Create a new bloom
19             Bloom bloom = new Bloom(new ServerFactory(), new DaemonFactory());
20
21             // Run the server(s)
22             bloom.Blossom();
23         }
24     }
25 }
```

Listing 43 - Minimalprojekt (ORCHID Server)

In Zeile 3 wird der Namespace `Orchid.Core.Configuration` eingebunden. Die darin enthaltenen Klassen ermöglichen es, auf das Konfigurationssystem zugreifen zu können. In Zeile 16 wird das Konfigurationssystem mit der Konfigurationsdatei *„`Settings/Orchid.ini`“* initialisiert (diese Datei muss vorhanden und lesbar sein und gültige Konfigurationselemente enthalten, vgl. 7.5). Es ist möglich, einen relativen (wie im obigen Beispiel) oder absoluten Pfad zu dieser Datei anzugeben.

Der einzige Konstruktor der Klasse `Bloom` erwartet als Parameter je eine Instanz einer Fabrikklasse für Server und Daemons (hier `ServerFactory` bzw. `DaemonFactory`). Ist eine neue Instanz erzeugt worden, kann der ORCHID Server mit dem Aufruf der Methode `Blossom()` *(dt.: „erblühen“)* der Instanz gestartet werden.

Dieses Minimalprojekt gibt keine Meldungen auf die Standardausgabe oder in eine Log-Datei aus, ist aber dennoch ein vollwertiger ORCHID Server. Im weiteren Verlauf wird, ausgehend von dem in diesem Abschnitt gezeigten Beispiel, die Weiterentwicklung des Minimalprojekts bis hin zur fertigen Referenzimplementierung beschrieben.

7.5 Konfiguration

Die Konfigurationsmöglichkeiten, die die Server-Komponente einem Benutzer bietet, sind sehr umfangreich und sollen in diesem Abschnitt erläutert werden.

Die Konfigurationsdatei *„Orchid.ini“* für die Server-Komponente befindet sich im Verzeichnis *Settings*, im Ordner *Orchid/Server/bin/* auf der DVD. Diese Datei ist die einzige Konfigurationsdatei, keine andere Datei wird verwendet.

Wie der Name der Datei bereits zeigt, handelt es sich bei *Orchid.ini* um eine sog. *„INI-Datei“*. Eine solche Datei enthält eine Sammlung von Schlüssel-Wert-Paaren (Key-Value-Pairs), die in Kategorien unterteilt sind (eine Kategorie wird definiert, indem ihr Name in eckigen Klammern angegeben wird). Eine INI-Datei bietet sich für die Konfiguration einer Server-Software an, da sie – im Gegensatz zu z.B. XML-Dateien – leicht von Menschen und Computern gelesen und editiert werden kann. Server-Betriebssysteme verzichten meist auf eine grafische Benutzeroberfläche, ihre Administration geschieht fast ausschließlich auf der Kommandozeile. Dies gilt auch für das Bearbeiten von Konfigurationsdateien. Eine INI-Datei lässt sich auf der Kommandozeile weitaus leichter bearbeiten, als dies bei einer XML-Datei der Fall ist.

Tabelle 89 listet die Kategorien in der Datei *Orchid.ini* auf und zeigt, welche Art von Einstellungen sich in der jeweiligen Kategorie befindet.

Kategorie	Art und Verwendung der enthaltenen Einstellungen
Dependencies	Einstellungen für das dynamische Typeloading
Server	Einstellungen für den Serverteil, der mit Clients kommuniziert
Restrictions	Sicherheitseinstellungen
Logging	Logging-Einstellungen
Database	Datenbankeinstellungen
Locale	Umgebungseinstellungen für die Ausgabe auf der Kommandozeile
Verbosity	Einstellungen bezüglich der Menge an Ausgaben auf der Kommandozeile
Debug	Einstellungen, die bei der Fehlersuche und –behebung nützlich sind

Tabelle 89 – Einstellungskategorien

Im Folgenden werden die einzelnen Einstellungen der o.g. Kategorien detailliert erläutert, da sie für den Einsatz des ORCHID Toolkits unverzichtbar sind. Zusätzlich liegt der begleitenden DVD (siehe Abschnitt 17) eine Konfigurationsdatei bei, die mit zahlreichen Kommentaren ausgestattet ist. Die Einstellungen sind dabei nach Zusammengehörigkeit und danach alphabetisch geordnet.

7.5.1 Kategorie „Dependencies“

Die Einstellungen in dieser Kategorie ermöglichen es, die gesamte Struktur der Server-Komponente zu verändern, indem Typ-Bindungen ausgetauscht werden können, ohne den Quellcode neu kompilieren zu müssen.

Einstellung	Bedeutung
ActorClass	Der volle Name der konkreten Implementierung der Klasse, die an *Orchid.Core.Data.AActor* gebunden werden soll.
ActorClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *ActorClass* genannt wird.
ArduinoClass	Der volle Name der konkreten Implementierung der

Einstellung	Bedeutung
	Klasse, die an `Orchid.Core.Data.AArduino` gebunden werden soll.
ArduinoClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *ArduinoClass* genannt wird.
BindingVerifierClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Data.ABindingVerifier` gebunden werden soll.
BindingVerifierClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *BindingVerifierClass* genannt wird.
CredentialsClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Data.ACredentials` gebunden werden soll.
CredentialsClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *CredentialsClass* genannt wird.
DatabaseAdapterClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Data.ADatabaseAdapter` gebunden werden soll.
DatabaseAdapterClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *DatabaseAdapterClass* genannt wird.
DataFactoryClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Data.ADataFactory` gebunden werden soll.
DataFactoryClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *DataFactoryClass* genannt wird.
GeolocationClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Data.AGeolocation` gebunden werden soll.
GeolocationClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *GeolocationClass* genannt wird.
HardwareDaemonClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Daemons.AHardwareDaemon` gebunden werden soll.
HardwareDaemonClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *HardwareDaemonClass* genannt wird.
HardwareFactoryClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Data.AHardwareFactory` gebunden werden soll.
HardwareFactoryClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *HardwareFactoryClass* genannt wird.

Einstellung	Bedeutung
HardwareServerClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.AHardwareServer` gebunden werden soll.
HardwareServerClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *HardwareServerClass* genannt wird.
InstructionClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Data.AInstruction` gebunden werden soll.
InstructionClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *InstructionClass* genannt wird.
ParserClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Parsers.AHttpRequestParser` gebunden werden soll.
ParserClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *ParserClass* genannt wird.
RespomderClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Responders.AHttpResponder` gebunden werden soll.
ResponderClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *ResponderClass* genannt wird.
SensorClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Data.ASensor` gebunden werden soll.
SensorClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *SensorClass* genannt wird.
SensorDataClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Data.ASensorData` gebunden werden soll.
SensorDataClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *SensorDataClass* genannt wird.
SoftwareDaemonClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Daemons.ASoftwareDaemon` gebunden werden soll.
SoftwareDaemonClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *SoftwareDaemonClass* genannt wird.
SoftwareServerClass	Der volle Name der konkreten Implementierung der Klasse, die an `Orchid.Core.Servers.ASoftwareServer` gebunden werden soll.
SoftwareServerClassAssemblyFile	Der relative oder absolute Pfad zu der Assemblydatei, die die Klasse enthält, die in → *SoftwareServerClass* genannt wird.

Einstellung	Bedeutung
FallBack	Bestimmt, ob zur Auflösung einer Typbindung im Fehlerfall auch auf die ausführende Assembly zurückgegriffen werden soll.
UseDefaultsOnError	Bestimmt, ob im Fehlerfall beim Laden eines Typs der fest kodierte Standardwert verwendet werden soll.

Tabelle 90 - Einstellungskategorie "Dependencies"

7.5.2 Kategorie „Server“

Diese Kategorie enthält Einstellungen, die verschiedene Aspekte des Serververhaltens beeinflussen.

Einstellung	Bedeutung
HardwareServerPort	Der Port, an dem eine Hardware-Server-Instanz auf eingehende Verbindungen wartet.
HardwareServerTimeout	Die Zeitspanne (in Sekunden), die eine Hardware-Server-Instanz abwartet, bevor die Beantwortung einer Anfrage abgebrochen wird.
ParserTimeout	Die Zeit (in Sekunden), die eine Parser-Instanz zum Parsen einer Anfrage maximal brauchen darf.
SoftwareServerPort	Der Port, an dem eine Software-Server-Instanz auf eingehende Verbindungen wartet.
SoftwareServerTimeout	Die Zeitspanne (in Sekunden), die eine Software-Server-Instanz abwartet, bevor die Beantwortung einer Anfrage abgebrochen wird.

Tabelle 91 - Einstellungskategorie "Server"

7.5.3 Kategorie „Restrictions“

In dieser Kategorie sind Einstellungen enthalten, die die Restriktionen (Sicherheits- und Stabilitätsfunktionen) der Serverkomponente bestimmen.

Einstellung	Bedeutung
AcceptedUserAgent	Der User-Agent, der vom Server akzeptiert wird.
VerifyUserAgent	Bestimmt, ob der Server den übertragenen User-Agent betrachtet oder nicht.
CredentialsEssential	Gibt an, ob Credentials verwendet werden sollen oder nicht.
DeveloperKey	Der Developer Key, der vom Server akzeptiert wird.
DeveloperKeyNeeded	Gibt an, ob der Server den Developer Key verwendet oder nicht.
RestrictToKnownIPs	Bestimmt, ob der Server nur Anfragen von ihm bekannten IP-Adressen entgegennimmt, oder ob jede IP-Adresse zugelassen ist.
VerifyDataStrings	Bestimmt, ob Datentyp-Strings in Sensordaten-Übertragungen an den Server auf Gültigkeit geprüft werden oder nicht.

Tabelle 92 - Einstellungskategorie "Restrictions"

7.5.4 Kategorie „Logging“

Die in dieser Kategorie enthaltenen Einstellungen bestimmen, wie sich das Logging in der Server-Komponente verhält.

Einstellung	Bedeutung
ConsoleLogEnabled	Bestimmt, ob Log-Meldungen auf Standardausgabe ausgegeben werden sollen oder nicht.
ConsoleLogLayout	Das Format von Log-Meldungen auf der Standardausgabe.
ConsoleMinLevel	Das Log-Level für Log-Meldungen auf der Standardausgabe. Mögliche Werte: `Debug`, `Warn`, `Error`, `Fatal`.
FileLogLayout	Das Format von Log-Meldungen, die in die Log-Datei geschrieben werden.
FileMinLevel	Das Log-Level für Log-Meldungen, die in die Log-Datei geschrieben werden. Mögliche Werte: `Debug`, `Warn`, `Error`, `Fatal`.
LogArchiveFileName	Das Format des Namens der Archivdatei für Log-Dateien (wen das Archivieren von Log-Dateien aktiviert ist, siehe auch → *LogFileRolling*).
LogFile	Der Pfad zur Log-Datei.
LogFileEnabled	Bestimmt, ob Log-Meldungen in die Log-Datei (→ *LogFile*) ausgegeben werden sollen oder nicht.
LogFileMaxSize	Die maximale Dateigröße einer Log-Datei in Bytes (wenn das Archivieren von Log-Dateien aktiviert ist, siehe auch → *LogFileRolling*).
LogFileRolling	Bestimmt, ob das Archivieren von Log-Dateien aktiviert ist oder nicht.
LoqSQL	Bestimmt, ob SQL-Abfragen als Log-Meldungen ausgegeben werden sollen oder nicht.

Tabelle 93 - Einstellungskategorie "Logging"

7.5.5 Kategorie „Database“

Enthält die zu verwendenden Datenbankparameter als Connection-String.

Einstellung	Bedeutung
DatabaseConnectionString	Der Connection-String, der zum Aufbau einer Datenbankverbindung verwendet wird.

Tabelle 94 - Einstellungskategorie "Database"

7.5.6 Kategorie „Locale“

Diese Kategorie enthält Einstellungen, die die Ausgabe der Server-Komponente abhängig von den Einstellungen des Betriebssystems beeinflussen.

Einstellung	Bedeutung
Culture	Die Kulturinformation[66] zur Formatierung von Ausgaben. Kann die System-Sprache überschreiben (um z.B. Zahlen

[66] http://msdn.microsoft.com/en-us/library/system.globalization.cultureinfo.aspx

Einstellung	Bedeutung
	mit Dezimalpunkt zu formatieren, obwohl ein englischsprachiges Betriebssystem verwendet wird).
UseEnvironmentLocale	Bestimmt, ob die unter → *Culture* angegebene Kulturinformation oder die System-Sprache zur Formatierung von Ausgaben verwendet werden soll.

Tabelle 95 - Einstellungskategorie "Locale"

7.5.7 Kategorie „Verbosity“

Die einzige Einstellung in dieser Kategorie bestimmt, ob eine Willkommensnachricht ausgegeben werden soll, wenn die Server-Komponente startet.

Einstellung	Bedeutung
PrintWelcomeMessage	Gibt den Inhalt der Datei „welcome.msg“ aus, wenn die Server-Komponente gestartet wird.

Tabelle 96 - Einstellungskategorie "Verbosity"

7.5.8 Kategorie „Debug“

Diese Kategorie enthält Einstellungen, die bei der Fehlersuche und –behebung hilfreich sein können.

Einstellung	Bedeutung
LogRequests	Bestimmt, ob Anfragen in die Log-Datei und/oder auf der Standardausgabe ausgegeben werden sollen oder nicht.
LogResponses	Bestimmt, ob die Antworten des Servers in die Log-Datei und/oder auf der Standardausgabe ausgegeben werden sollen oder nicht.

Tabelle 97 - Einstellungskategorie "Debug"

7.6 Aktivierung des Logging-Systems

Das in 7.4 gezeigte Minimalprojekt nutzt die Möglichkeiten, die das Logging-System bietet, nicht. Entsprechend werden keine Meldungen auf der Standardausgabe oder in eine Log-Datei ausgegeben. Im Folgenden wird beschrieben, wie das Logging-System aktiviert werden kann.

Ein Entwickler hat zwei Möglichkeiten, Ausgaben der Server-Komponente zu ermöglichen. Wird dem Projekt eine NLog-Konfigurationsdatei hinzugefügt (vgl. 5.5.1), werden die Logging-Einstellungen aus der Konfigurationsdatei nicht verwendet (um die Verwendung des Logging-Systems möglichst flexibel zu gestalten, wird dessen Konfiguration nicht über eine Klasse aus `Orchid.Core.dll` übernommen, sondern muss von einem Entwickler stets selbst durchgeführt werden). Die zweite Möglichkeit, nämlich die Konfiguration aus dem Code der Anwendung heraus, wird in diesem Abschnitt beschrieben. In einem Beispiel wird dazu gezeigt, wie Ausgaben auf der Standardausgabe aktiviert werden können; die Referenzimplementierung eines ORCHID Servers, die auf der beliegenden DVD enthalten ist, ist darüber hinaus in der Lage, in Log-Dateien zu schreiben. Die Aktivierung der Ausgabe von Log-Meldungen in Dateien verhält sich analog zu der in diesem Abschnitt gezeigten Vorgehensweise.

Im nachfolgenden Listing 44 wird eine neue Methode `InitializeLogging()` definiert, die die Einstellungen für das Logging-System (es werden nur die Einstellungen für die Standardausgabe betrachtet) aus der Konfigurationsdatei ausliest und NLog konfiguriert. Der Aufruf dieser Methode wird der `Main()`-Methode hinzugefügt.

```
01 using System;
02 using System.IO;
03 using NLog;
04 using NLog.Config;
05 using NLog.Targets;
06 using Orchid.Core;
07 using Orchid.Core.Configuration;
08 using Orchid.Core.Servers;
09 using Orchid.Core.Daemons;
10
11 namespace OrchidReferenceServer
12 {
13     public static class MainClass
14     {
15         private static Config _cfg = Config.Instance;
16
17         public static void Main (string[] args)
18         {
19             string cfgFile = "Settings/Orchid.ini";
20
21             // Initialize configuration system
22             _cfg.Initialize(new FileInfo(cfgFile));
23
24             // Initialize Logging-System
25             InitializeLogging();
26
27             // Create a new bloom
28             Bloom bloom = new Bloom(new ServerFactory(), new DaemonFactory());
29
30             // Run the server(s)
31             bloom.Blossom();
32         }
33
34         private static void InitializeLogging()
35         {
36             bool enabled = _cfg.GetBool("Logging", "ConsoleLogEnabled", true);
37
38             LoggingConfiguration conf = new LoggingConfiguration();
39
40             if(enabled)
41             {
42                 ConsoleTarget t = new ConsoleTarget();
43                 t.Layout = _cfg.Get("Logging", "ConsoleLogLayout", "$message");
44                 conf.AddTarget("console", t);
45
46                 LogLevel l = LogLevel.FromString(
                       _cfg.Get("Logging", "ConsoleMinLevel", "Info");
47
48                 LoggingRule rule = new LoggingRule("*", l, t);
49
50                 conf.LoggingRules.Add(rule);
51             }
52
53             LogManager.Configuration = conf;
54         }
55     }
56 }
```

Listing 44 - Initialisierung des Logging-Systems

In einer nicht überwachten Umgebung, also z.B. auf einem entfernten Server, sollte der in Listing 44 gezeigte Code nicht ohne weitere Anpassungen verwendet werden, da u.a. Fehler nicht abgefangen werden, sondern zu einem Absturz des Programms führen können.

Um den Server nach einem problematischen Fehler neu starten zu lassen, können die Erzeugung der Instanz der Klasse *Bloom* und der Aufruf der Methode *Blossom()* in eine endlose *while*-Schleife integriert werden.

Das folgende Listing zeigt die o.g. Verbesserungen an einem Beispiel. Das veränderte Programm erzeugt im Fehlerfall immer wieder eine neue Instanz der Klasse *Bloom* und startet mit dem Aufruf von *Blossom()* die Server-Instanzen neu.

```
01 namespace OrchidReferenceServer
02 {
03     public static class MainClass
04     {
05         public static void Main (string[] args)
06         {
07             string cfgFile = "Settings/Orchid.ini";
08 
09             // Initialize configuration system
10             Config.Instance.Initialize(new FileInfo(cfgFile));
11 
12             while(true)
13             {
14                 try
15                 {
16                     // Create a new bloom
17                     Bloom bloom = new Bloom(new ServerFactory(),
                                                new DaemonFactory());
18                     // Run the server(s)
19                     bloom.Blossom();
20                 }
21                 catch (OrchidMySqlException oEx)
22                 {
23                     // Output error message
24                 }
25                 catch (OrchidException orEx)
26                 {
27                     //Output error message
28                 }
29                 catch (Exception ex)
30                 {
31                     //Output error message
32                 }
33             }
34         }
35     }
36 }
```

Listing 45 - Verbessertes Minimalprojekt (ORCHID Server)

7.7 Veränderung und Erweiterung der Server-Komponente

Im Folgenden wird detailliert beschrieben, wie die Server-Komponente verändert und erweitert werden kann. Zu diesem Zweck wird als Veränderung eine Daemon-Klasse entwickelt, die TRACE-Anfragen eines ORCHID Clients anders behandelt, als dies in der Standardklasse `OrchidHardwareDaemon` der Fall ist. Zusätzlich ist die neue Klasse in der Lage, HEAD-Anfragen eines Clients entgegenzunehmen, was eine Erweiterung der Möglichkeiten der Server-Komponente darstellt.

Aufgrund der Länge des im Rahmen dieses Abschnitts entwickelten Quellcodes werden stets nur Code-Auszüge gezeigt. Der vollständige Code der Erweiterung befindet sich auf dem dieser Arbeit beiliegenden Datenträger.

7.7.1 Veränderung

Um eine neue Implementierung einer existierenden Klasse der Server-Komponente zu entwickeln, sind mehrere Schritte notwendig.

Zunächst muss ein Projekt erstellt werden, das die neue Klasse bereitstellt. Es ist zwar möglich, die Klasse dem ursprünglichen Projekt der Server-Komponente hinzuzufügen, es wird jedoch empfohlen, sie über eine eigene Assembly bereitzustellen. Wie in 7.3 beschrieben, muss die Assembly `Orchid.Core.dll` dem neuen Projekt hinzugefügt werden, damit die darin enthaltenen Klassen aus dem neuen Projekt heraus verwendet werden können.

Ist die Server-Komponente referenziert, muss die neu zu implementierende Klasse definiert und entweder von einer abstrakten Basisklasse oder einer konkreten Implementierung abgeleitet werden. Wird sie von einer abstrakten Klasse abgeleitet, müssen alle Methoden entsprechend selbst implementiert werden; wird von einer konkreten Implementierung abgeleitet, können punktuelle Änderungen vorgenommen werden und die übrigen Abläufe unverändert bestehen bleiben.

Im letzten Schritt muss der Server-Komponente über die Konfigurationsdatei mitgeteilt werden, dass die neue Implementierung anstelle der existierenden verwendet werden soll und in welcher Assembly die entsprechende Klasse zu finden ist.

Das folgende Beispiel zur Implementierung einer neuen Daemon-Klasse, verdeutlicht die o.g. Schritte. Zunächst wird eine neue Klasse namens `CustomHardwareDaemon` hinzugefügt, die von der konkreten Implementierung `OrchidHardwareDaemon` abgeleitet wird. Listing 46 zeigt das Grundgerüst dieser Klasse.

```
01 public class CustomHardwareDaemon : OrchidHardwareDaemon
02 {
03 }
```

Listing 46 - Grundgerüst "CustomHardwareDaemon"

Da die Klasse `OrchidHardwareDaemon` nur über einen öffentlichen Konstruktor verfügt, der mehrere Parameter erfordert, muss auch der neuen Klasse ein solcher Konstruktor hinzugefügt werden. Dieses Verfahren stellt sicher, dass das dynamische Typeloading (vgl. 6.3.2) alle Abhängigkeiten einer Instanz der Klasse korrekt auflösen kann.

Der Konstruktor wird wie im folgenden Listing 47 gezeigt definiert.

```
01 public CustomHardwareDaemon(ADatabaseAdapter database,
                               AHardwareServer server,
                               AHttpRequestParser parser,
                               ABindingVerifier verifier,
                               AHttpResponder responder,
                               AHardwareFactory hardwareFactory,
                               ADataFactory dataFactory) : base(database,
                                                                server,
                                                                parser,
                                                                verifier,
                                                                responder,
                                                                hardwareFactory,
                                                                dataFactory)
```

Listing 47 - Konstruktor "CustomHardwareDaemon"

Die neue Klasse ist durch die Regeln der Objektorientierung bereits jetzt als neue Daemon-Klasse einsetzbar, da sie alle Eigenschaften und Methoden der abgeleiteten Klasse *OrchidHardwareDaemon* erbt und vom Typeloading-System korrekt initialisiert werden kann. Sie stellt jedoch bisher keine Veränderung dar.

Um die Behandlung von Anfragen eines Clients zu verändern, muss die Methode *HandleRequest()* überschrieben werden, die jede Daemon-Klasse anbietet.

```
01 public override void HandleRequest()
02 {
03     BufferedStream bs = new BufferedStream(this.Socket.GetStream());
04
05     this.SetRequest(this.Parser.ParseRequest(stream));
06
07     if(this.Request == null || this.Request.count == 0) { return; }
08
09     switch(this.Request.RequestType)
10     {
11         case HttpRequestType.Get: { this.HandleGETRequest(); break; }
12         case HttpRequestType.Put: { this.HandlePUTRequest(); break; }
13         case HttpRequestType.Trace: { this.HandleTRACERequest(); break; }
14         default: { this.HandleUnknownRequestType(); break; }
15     }
16 }
17
18 protected internal new void HandleTRACERequest()
19 {
20     // New handling goes in here
21 }
```

Listing 48 - Überschreiben von "HandleRequest"

Das obige Listing 48 zeigt, wie die Methode *HandleRequest()* überschrieben werden kann.

Zunächst muss die Anfrage, die eine Instanz einer Server-Klasse an den Daemon weiterleitet, aus einem Datenstrom ausgelesen werden. Dieser Datenstrom wird in Zeile 3 aus dem Socket, über den die aktuelle Daemon-Instanz mit einem Client verbunden ist, ermittelt und in eine Instanz der Klasse *BufferedStream* gekapselt. Eine solche Instanz wird von der Methode *ParseRequest()* erwartet, die von der Klasse *AHttpRequestParser* angeboten wird und den Datenstrom in eine komfortablere Datenstruktur vom Typ *AHttpRequest* überführt. In Zeile 5 wird der aktuelle Request, den die Instanz der neuen Daemon-Klasse verarbeitet, auf das Ergebnis der Methode *ParseRequest()*

gesetzt. Die Eigenschaft `Request` enthält danach die aufbereitete HTTP-Anfrage des Clients (oder ist leer). Ist `Request` leer, wird in Zeile 7 die Verarbeitung abgebrochen.

Sind Daten in `Request` vorhanden, wird über die HTTP-Methode der aufbereiteten Anfrage (`this.Request.RequestType`) ermittelt, welche Methode zur Verarbeitung der Anfrage aufgerufen werden soll. Für dieses Beispiel ist insbesondere Zeile 13 interessant, da die Behandlung von TRACE-Anfragen verändert werden soll.

Um eine TRACE-Anfrage eines Clients zu verarbeiten, wird die Methode `HandleTRACERequest()` aufgerufen. Solange sie nicht in der neuen Klasse überschrieben wird, handelt es sich dabei um die ursprüngliche Methode aus `OrchidHardwareDaemon`. In Zeile 18 wird jedoch eine neue Methode mit dem gleichen Namen deklariert, weshalb der Aufruf in Zeile 13 nun die Methode der neuen Klasse aufruft und die ursprüngliche Behandlung der Anfrage nicht mehr stattfindet.

Die Behandlung der TRACE-Anfrage muss nun – dem in 5.8 beschriebenen Protokoll folgend – implementiert werden. Der Einfachheit halber wird für dieses Beispiel auf die Gültigkeitsprüfungen und die Analyse der Anfrage verzichtet und stattdessen auf jede TRACE-Anfrage eine Antwort mit HTTP-Code 418 (*„I'm a Teapot"*, vgl. [25], Abschnitt 2.3.2) gesendet, wie im folgenden Listing dargestellt ist.

```
01 protected internal new void HandleTRACERequest()
02 {
03     this.Respond(HttpProtocolHelper.HttpResponseHeaderTeapot,
                    HttpProtocolHelper.HttpResponseTextTeapot);
04 }
```

Listing 49 - Methode zur Behandlung von TRACE-Requests

Nach diesem letzten Schritt ist die neue Daemon-Klasse einsatzbereit und kann der Server-Komponente über die Konfigurationsdatei bekannt gemacht werden. Für das folgende Konfigurationsbeispiel wird angenommen, dass die neue Klasse in der Assembly `CustomDaemon.dll` enthalten ist, die sich im gleichen Verzeichnis befindet wie die Assembly `Orchid.Core.dll`. Darüber hinaus wird angenommen, dass sie sich im Namespace `Custom.Daemons` befindet. Weichen die tatsächlichen Namen ab, müssen die im Folgenden gezeigten Einstellungen entsprechend verändert werden.

Wie aus Abschnitt 7.5.1 ersichtlich, wird der Server-Komponente über das Konfigurationselement `HardwareDaemonClass` mitgeteilt, welche Klasse zur Behandlung von Anfragen durch ORCHID Clients verwendet werden soll. Über das Element `HardwareDaemonClassAssemblyFile` wird festgelegt, in welcher Assembly sich diese Klasse befindet.

Um die neue Klasse zu verwenden, muss die Konfiguration entsprechend folgendem Listing angepasst werden.

```
01 HardwareDaemonClass = „Custom.Daemons.CustomHardwareDaemon"
02 HardwareDaemonClassAssemblyFile = „CustomDaemon.dll"
```

Listing 50 - Anpassung der Konfigurationsdatei

Wird nun ein ORCHID Server mit der neuen Konfiguration gestartet, werden alle Anfragen, die durch ORCHID Clients gesendet werden, durch die neue Klasse `CustomHardwareDaemon` behandelt und TRACE-Anfragen stets mit HTTP-Code 418 (*„I'm a Teapot"*) beantwortet.

7.7.2 Erweiterung

Die im vorherigen Abschnitt entwickelte Klasse `CustomHardwareDaemon` verändert das Verhalten der Server-Komponente, fügt ihr aber keine neue Funktionalität hinzu. Dieser zweite Schritt wird in diesem Abschnitt beschrieben, indem die Klasse in die Lage versetzt wird, HEAD-Anfragen von ORCHID Clients zu behandeln.

Es wurde bereits gezeigt, wie die Unterscheidung von verschiedenen Request-Typen anhand deren HTTP-Methode durchgeführt werden kann. Entsprechend muss zur Behandlung eines weiteren Typs die dafür zu verwendende Methode genannt und implementiert werden, wie das folgende Listing zeigt.

```
01 public override void HandleRequest()
02 {
03     BufferedStream bs = new BufferedStream(this.Socket.GetStream());
04
05     this.SetRequest(this.Parser.ParseRequest(stream));
06
07     if(this.Request == null || this.Request.count == 0) { return; }
08
09     switch(this.Request.RequestType)
10     {
11         case HttpRequestType.Get: { this.HandleGETRequest(); break; }
12         case HttpRequestType.Put: { this.HandlePUTRequest(); break; }
13         case HttpRequestType.Trace: { this.HandleTRACERequest(); break; }
14         case HttpRequestType.Head: { this.HandleHEADRequest(); break; }
15         default: { this.HandleUnknownRequestType(); break; }
16     }
17 }
18
19 protected internal new void HandleHEADRequest()
20 {
21     // New handling goes in here
22 }
```

Listing 51 - Veränderung der Methode zur Anfragebehandlung

Die Art der Behandlung neuer Request-Arten ist vom Entwickler der neuen Klasse frei definierbar, muss sich aber an die Regeln des in 5.8 festgelegten Protokolls halten, damit es nicht zu Problemen mit anderen Bestandteilen des ORCHID Toolkits kommt.

Ein Beispiel für eine nicht protokollkonforme Behandlung von HEAD-Requests kann Listing 52 entnommen werden.

```
01 protected internal new void HandleHEADRequest()
02 {
03     using(FileStream fs = new FileStream("out.txt",
                                            FileMode.OpenOrCreate,
                                            FileAccess.Write))
04     {
05         using(StreamWriter sw = new StreamWriter(fs))
06         {
07             sw.WriteLine("Head request received!");
08         }
09     }
10
11 }
```

Listing 52 – Ungültige Methode zur Behandlung von HEAD-Requests

Die Verarbeitung des HEAD-Requests ist deshalb ungültig, weil die in Listing 52 gezeigte Methode die Anfrage nicht wie im Protokoll festgelegt beantwortet. Wird eine Beantwortung der Anfrage hinzugefügt, ist die Verarbeitung gemäß Protokoll gültig. Listing 53 zeigt abschließend die gültige Methode.

```
01 protected internal new void HandleHEADRequest()
02 {
03     long systemTicks = Environment.TickCount;
04     this.Respond(HttpProtocolHelper.HttpResponseHeaderOk,
                    systemTicks.ToString());
05
06 }
```

Listing 53 - Gültige Methode zur Behandlung von HEAD-Requests

Diese Art der Verarbeitung ist gültig und sendet einem Client als Antwort die Anzahl der Millisekunden, die vergangen sind, seit das System (in diesem Fall der Server) gestartet wurde.

8 Handbuch Client-Bibliothek

Dieser Abschnitt bildet das Handbuch zur Verwendung der Client-Bibliothek. Zunächst wird auf die Voraussetzungen eingegangen, die ein System, auf dem die Bibliothek eingesetzt werden soll, erfüllen muss. Danach werden Installation und Verwendung der Bibliothek erläutert und Codebeispiele gegeben.

8.1 Systemvoraussetzungen

Die Client-Bibliothek stellt keine gesonderten Anforderungen an ein System. Sie kann auf allen Systemen verwendet werden, für die eine Arduino Entwicklungsumgebung verfügbar ist. Zur Verwendung der Bibliothek ist ein Arduino Microcontroller-Board mit Ethernet-Shield (oder ein kompatibles Microcontroller-Board) notwendig, das mindestens die in 4.3.2.2 beschriebenen Anforderungen erfüllt.

8.2 Installation

Um die Client-Bibliothek verwenden zu können, muss auf dem Entwicklungssystem eine Arduino IDE installiert sein. Die Installation der IDE für Microsoft Windows und Linux wird im Folgenden beschrieben; eine vollständige Anleitung zur Installation unter Windows, Linux und MacOS kann der Website des Herstellers entnommen werden[67].

8.2.1 Installation der Arduino IDE

Die Arduino IDE kann kostenlos für Microsoft Windows, Linux und MacOS heruntergeladen werden[68]. Nachdem die für das Entwicklungssystem passende Version der IDE von der Herstellerwebsite heruntergeladen und entpackt wurde, kann die IDE ohne weitere Installation gestartet werden (erfordert Java). Unter Microsoft Windows muss dazu das Programm „`arduino.exe`“ gestartet werden, die Distribution für Linux enthält ein Shellscript namens „`arduino`“. Nach dem Start öffnet sich das Hauptfenster der Arduino IDE, das in Abbildung 42 zu sehen ist.

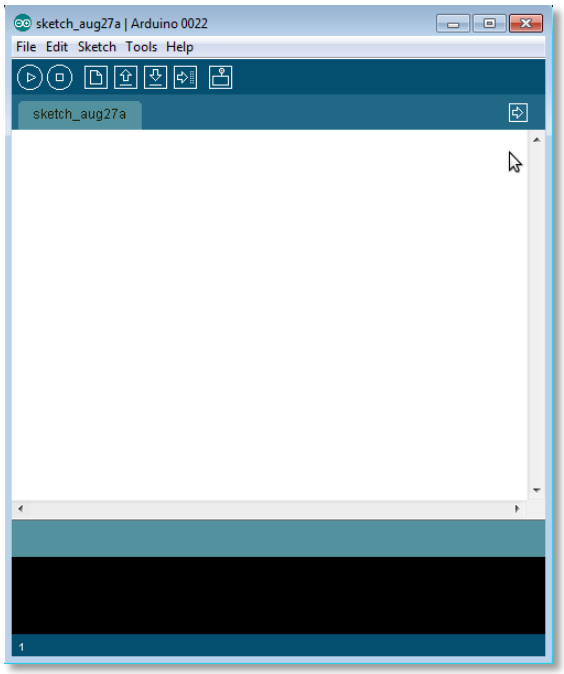

Abbildung 42 - Hauptfenster der Arduino IDE

[67] http://arduino.cc/en/Guide/HomePage
[68] http://arduino.cc/en/Main/Software

8.2.2 Installation der Client-Bibliothek

Die Arduino IDE erwartet, dass Bibliotheken, die in Sketches genutzt werden sollen, in einem bestimmten Verzeichnis abgelegt werden.

Unter Microsoft Windows befindet sich dieses Verzeichnis im Order *„Arduino“*, der im Dokumente-Ordner des aktuellen Benutzers zu finden ist. Listing 54 zeigt, wie dieser Pfad unabhängig vom Benutzernamen gefunden werden kann (Windows 7, Eingabe in die Adresszeile des Windows Explorers).

```
%HOMEPATH%\Documents\Arduino
```

Listing 54 - Pfad zum Sketch-Verzeichnis unter Microsoft Windows

Der Ordner *„Arduino“* enthält einen weiteren Ordner namens *„libraries“*. Ist dieser zweite Ordner nicht vorhanden, muss er erzeugt werden.

Unter Linux befindet sich das Verzeichnis *„libraries“* im Ordner *„sketchbook“* im Home-Verzeichnis des angemeldeten Benutzers. Ist es nicht vorhanden, muss es erzeugt werden. Das folgende Listing zeigt, wie das Verzeichnis unter Linux unabhängig vom Benutzernamen gefunden werden kann.

```
cd ~/sketchbook
```

Listing 55 - Pfad zum Sketch-Verzeichnis unter Linux

Die dieser Arbeit beiliegende DVD beinhaltet die Client-Bibliothek im Ordner *„Orchid/Client/lib“*. Dieser Ordner enthält einen weiteren Ordner namens *„Orchid“*, der in das jeweilige *„libraries“*-Verzeichnis kopiert werden muss, um die Client-Bibliothek in eigenen Sketches nutzen zu können.

Der Erfolg der Installation der Client-Bibliothek kann über die Arduino IDE geprüft werden. Über das Menü *„Sketch → Import Library“* können in der IDE externe Bibliotheken eingebunden werden. Erscheint die Bibliothek *„Orchid“* in der Liste der verfügbaren Bibliotheken, war die Installation erfolgreich. Abbildung 43 zeigt das entsprechende Menü nach erfolgreicher Installation.

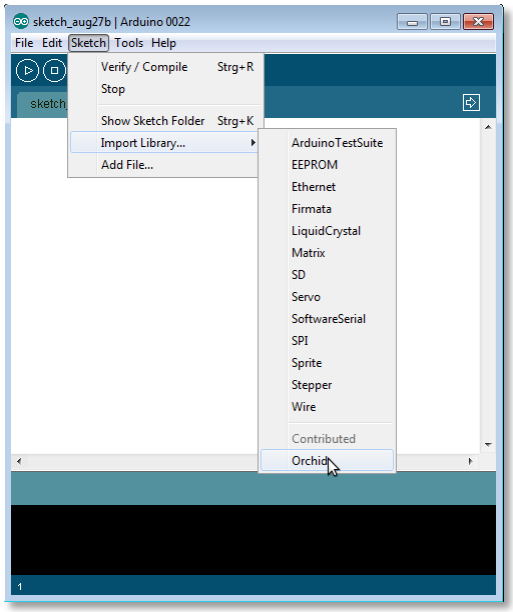

Abbildung 43 - Bibliotheksimport in der Arduino IDE

8.3 Referenzierung

Die Client-Bibliothek besteht aus mehreren Dateien, die in der folgenden Tabelle aufgeführt sind.

Datei	Beschreibung
`Orchid2.cpp`	Die Hauptdatei der Bibliothek.
`Orchid2.h`	Die zu Orchid2.cpp gehörende Header-Datei.
`OrchidInstruction.h`	Die Klasse, die Instruktionen kapselt.
`Strings.cpp`	Hilfsbibliothek für String-Operationen.
`Strings.h`	Die zu Strings.cpp gehörende Header-Datei.

Tabelle 98 - Quelldateien der Client-Bibliothek

Um die Bibliothek in einem Sketch nutzen zu können, müssen die in der obigen Tabelle gezeigten Header-Dateien (`Orchid2.h`, `OrchidInstruction.h` und `Strings.h`) in den Sketch eingebunden werden.

Die Einbindung kann entweder von Hand oder über den Menübefehl *„Sketch → Import Library“* vorgenommen werden. In beiden Fällen werden die in Listing 56 gezeigten Zeilen am Anfang des Sketchs eingefügt.

```
01 #include <Orchid2.h>
02 #include <OrchidInstruction.h>
03 #include <Strings.h>
```

Listing 56 - Referenzierung der Client-Bibliothek

Das Einbinden der Header-Datei *„`Strings.h`“* ist optional und nur notwendig, wenn die darin enthaltenen Funktionen im einbindenden Sketch genutzt werden sollen.

8.4 Codebeispiele

Dieser Teil des Handbuchs zur Client-Bibliothek erläutert, wie die eingebundene Bibliothek in eigenen Sketches verwendet werden kann, um den ausführenden Arduino Microcontroller zu einem ORCHID Client zu machen.

Jedes hier genannte Beispiel führt einen praktischen Verwendungszweck für die beschriebenen Aktionen an (Ausnahme ist das erste Beispiel, das lediglich die Initialisierung abdeckt) und befindet sich auf der dieser Arbeit beiliegenden DVD.

8.4.1 Initialisierung

Dieses Beispiel zeigt, wie die Client-Bibliothek eingebunden und initialisiert werden kann. Alle weiteren Beispiele basieren auf diesem, der Initialisierungscode wird deshalb in den folgenden Beispielen nicht wiederholt.

Zunächst wird über *„File → New“* ein neuer Sketch erzeugt oder ein bestehender Sketch in die Arduino IDE geladen. Jeder gültige Arduino-Sketch verfügt mindestens über die Methoden `setup()` und `loop()`. Sind diese Methoden im neuen oder geladenen Sketch nicht vorhanden, müssen sie erzeugt werden.

Der Aufbau des Sketchs ähnelt nun dem im folgenden Listing gezeigten.

```
01 // Example sketch No. 1
02
03 void setup() { }
04
05 void loop()  { }
```

Listing 57 - Grundgerüst eines Sketchs

Im nächsten Schritt muss die Client-Bibliothek des ORCHID Toolkits in den Sketch eingebunden werden. Dies geschieht entweder manuell oder über die Menüs der Arduino IDE. Nach der Einbindung hat sich der Code des Sketchs entsprechend Listing 58 verändert.

```
01 #include <Orchid2.h>
02 #include <OrchidInstruction.h>
03 #include <Strings.h>
04
05 void setup()
06 {
07 }
08
09 void loop()
10 {
11 }
```

Listing 58 - Einbindung der Client-Bibliothek

Nachdem die Client-Bibliothek eingebunden wurde, kann ein globales Objekt vom Typ `Orchid` erzeugt werden, über das der Sketch mit einem ORCHID Server kommunizieren kann.

Die Klasse `Orchid` verfügt über einen einzigen Konstruktor, der eine Reihe von Parametern erwartet. Die meisten dieser Parameter werden genutzt, um Objekte vom Typ `Client` aus der mit der Arduino IDE ausgelieferten Ethernet-Bibliothek zu erzeugen. Solche Objekte ermöglichen es einem Arduino, mit einem Server im lokalen Netzwerk oder im Internet zu kommunizieren. Ein Objekt vom Typ `Orchid` wird wie in Listing 59 gezeigt erzeugt.

```
01 Orchid orchid(mac, routerip, subnet, ip, port, server, arduino_id, devkey);
```

Listing 59 - Erzeugung eines Orchid-Objekts (1)

Tabelle 99 erläutert die im obigen Listing gezeigten Parameter.

Parameter	Funktion
mac	MAC-Adresse des Arduino Microcontrollers (frei wählbar).
routerip	IP-Adresse des zu verwendenden Gateways.
subnet	Zu verwendende Subnetzmaske.
ip	Die IP-Adresse, die dem Arduino zugewiesen wird.
port	Der Port des Servers, zu dem sich der Client verbinden soll.
server	Die IP-Adresse des ORCHID Servers, mit dem sich der Client verbinden soll.
arduino_id	Die GUID des Clients (vgl. 6.3.1).
devkey	Der Developer Key, der verwendet werden soll (vgl. 6.3.3.2).

Tabelle 99 - Konstruktorparameter der Klasse Orchid

Die Ethernet-Bibliothek erlaubt es einem Entwickler, die MAC-Adresse des Arduinos frei zu wählen. Der Konstruktor erwartet im entsprechenden Parameter ein Array von Bytes (Länge 6) in hexadezimaler Notation.

Die Parameter *routerip*, *subnet*, *ip* und *server* müssen mit Arrays von Bytes der Länge 4 in hexadezimaler Notation gefüllt und entsprechend der vorhandenen Konfiguration gewählt werden (IP-Adresse, Subnetzmaske und Gateway müssen den Anforderungen des vorhandenen Netzwerks entsprechen).

Das folgende Listing 60 zeigt die Initialisierung eines Objekts vom Typ *Orchid* mit allen notwendigen Werten.

```
01 #include <Orchid2.h>
02 #include <OrchidInstruction.h>
03 #include <Strings.h>
04
05 // Example sketch
06
07 byte mac[] = { 0x12, 0x23, 0x34, 0x45, 0x56, 0x67 };
08 byte gateway[] = { 192, 168, 1, 1 };
09 byte subnet[] = { 255, 255, 255, 255 };
10 byte ip[] = { 192, 168, 1, 150 };
11 byte server[] = { 192, 168, 1, 140 };
12 char* client_id = "469abf7f-c733-4a2e-b469-7baa15d77673";
13 int port = 29100;
14 char* dev_key = "8178b0492fad1e60f59706d84146f17b4bf1afeaba9c2b7dd3db102450cb6b9c18347806"
15
16 Orchid orchid(mac, gateway, subnet, ip, port, server, client_id, dev_key);
17
18 void setup()
19 {
20 }
21
22 void loop()
23 {
24 }
```

Listing 60 - Erzeugung eines Orchid-Objekts (2)

Die im oben gezeigten Beispiel verwendeten Werte orientieren sich an möglichst häufig verwendeten Werten in einem lokalen Netzwerk, müssen aber ggf. der tatsächlichen Situation angepasst werden.

In Zeile 16 wird die eigentliche Instanz des ORCHID Clients erzeugt. Die Instanz ist jedoch noch nicht einsatzbereit, da die Erzeugung die Ethernet-Bibliothek nicht automatisch initialisiert. Zu diesem Zweck stellt die Klasse *Orchid* die Methode *reset()* bereit, die die Initialisierung übernimmt. Diese Methode muss vor der ersten Verwendung der Instanz aufgerufen werden; der Code des Sketchs verändert sich also wie in Listing 61 zu sehen.

```
#include <Orchid2.h>
#include <OrchidInstruction.h>
#include <Strings.h>

byte mac[] = { 0x12, 0x23, 0x34, 0x45, 0x56, 0x67 };
byte gateway[] = { 192, 168, 1, 1 };
byte subnet[] = { 255, 255, 255, 255 };
byte ip[] = { 192, 168, 1, 150 };
byte server[] = { 192, 168, 1, 140 };
char* client_id = "469abf7f-c733-4a2e-b469-7baa15d77673";
int port = 29100;
char* dev_key = "8178b0492fad1e60f59706d84146f17b4bf1afeaba9c2b7dd3db102450cb6b9c18347806";

Orchid orchid(mac, gateway, subnet, ip, port, server, client_id, dev_key);

void setup()
{
    orchid.reset();
}

void loop() { }
```

Listing 61 - Initialisierung eines Orchid-Objekts

Nach dem Aufruf von `orchid.reset()` kann die Instanz im weiteren Verlauf des Sketchs verwendet werden. Die Verwendung wird in den folgenden Beispielen demonstriert.

8.4.2 Senden von Heartbeats

Für dieses Beispiel gilt das folgende Szenario: der Zustand eines Arduinos soll aus der Ferne überwacht werden. Dabei ist lediglich wichtig, dass der Microcontroller mit Strom versorgt und mit dem Internet verbunden ist. Zu diesem Zweck kann der Arduino über eine Client-Instanz einen *Heartbeat* an einen ORCHID Server senden, wie im folgenden Listing gezeigt wird.

```
void loop()
{
    orchid.heartbeat();
    delay(10000);
}
```

Listing 62 - Senden eines Heartbeats

Wird der im obigen Listing gezeigte Sketch (zusammen mit dem in 8.4.1 behandelten Initialisierungscode) von einem Arduino ausgeführt, sendet dieser alle zehn Sekunden einen *Heartbeat* an den angegebenen ORCHID Server.

Dieses Beispiel zeigt deutlich, dass die Client-Bibliothek es einem Entwickler sehr leicht macht, innerhalb eines Sketches mit einem ORCHID Server zu kommunizieren – für die einfachste Form der Kommunikation ist lediglich eine einzige Zeile Code notwendig.

8.4.3 Hochladen von Sensordaten

Möchte ein Entwickler in einer Software regelmäßig Sensordaten abrufen, die von einem an einen Arduino angeschlossenen Sensor gemessen werden, müssen diese auf einen ORCHID Server hochgeladen werden. Die Client-Bibliothek ermöglicht das Hochladen von Werten verschiedenster Datentypen; dieses Beispiel zeigt, wie ein Integer-Wert, der von einem Sensor gemessen wurde, hochgeladen wird.

Zusätzlich zu den in 8.4.1 gezeigten Initialisierungsdaten muss für dieses Beispiel auch eine eindeutige ID für den Sensor, für den Daten an den Server geschickt werden sollen, bekannt sein. Eine zusätzliche globale Variable erfüllt diesen Zweck, wie im folgenden Listing 63 zu sehen ist (Definition und Initialisierung der Sensor-ID in Zeile 14).

```
01 #include <Orchid2.h>
02 #include <OrchidInstruction.h>
03 #include <Strings.h>
04
05 byte mac[] = { 0x12, 0x23, 0x34, 0x45, 0x56, 0x67 };
06 byte gateway[] = { 192, 168, 1, 1 };
07 byte subnet[] = { 255, 255, 255, 255 };
08 byte ip[] = { 192, 168, 1, 150 };
09 byte server[] = { 192, 168, 1, 140 };
10 char* client_id = "469abf7f-c733-4a2e-b469-7baa15d77673";
11 int port = 29100;
12 char* dev_key = "8178b0492fad1e60f59706d84146f17b4bf1afeaba9c2b7dd3db102450cb6b9c18347806";
13
14 char* sensor_id = "531a3bbe-2bec-4bff-a4c3-c60645943f6d";
15
16 Orchid orchid(mac, gateway, subnet, ip, port, server, client_id, dev_key);
17
18 void setup()
19 {
20     orchid.reset();
21 }
22
23 void loop()
24 {
26     int senseVal = analogRead(sensorPin);
27     orchid.push(sensor_id, senseVal);
28     delay(10000);
29 }
```

Listing 63 - Hochladen von Sensordaten

In Zeile 26 des obigen Beispiels werden die Sensordaten gemessen (es wird davon ausgegangen, dass ein Sensor an dem analogen Pin des Arduinos angeschlossen ist, der den Index *sensorPin* trägt). Zeile 27 enthält den Aufruf der Methode *push(char*, int)* der Klasse *Orchid*, die es ermöglicht, Sensordaten an einen Server zu übermitteln. Diese Methode verfügt über zahlreiche Überladungen, so dass auch Werte anderer Datentypen als Integer (wie im obigen Beispiel) hochgeladen werden können.

Wie auch im vorherigen Beispiel wird nach dem Hochladen der Sensordaten zehn Sekunden gewartet, bevor die *loop()*-Methode erneut ausgeführt wird. Als Ergebnis wird der aktuelle Sensorwert alle zehn Sekunden auf den Server geladen, die entsprechende Größe (z.B. Luftfeuchtigkeit oder Raumtemperatur) kann in regelmäßigen Abständen überwacht werden.

8.4.4 Abrufen von Instruktionen

In diesem Beispiel wird davon ausgegangen, dass ein Entwickler über eine Software ein Gerät steuern möchte, das mit dem Arduino verbunden ist, der den in diesem Beispiel gezeigten Sketch ausführt.

Um diese Aufgabe zu bewältigen, muss der Entwickler eine entsprechende Instruktion auf dem ORCHID Server hinterlegen, mit dem sich der Client verbindet. Der Client ruft diese Instruktion ab und der Entwickler kann im Code des Sketchs darauf reagieren.

Zusätzlich zu dem in 8.4.1 gezeigten Initialisierungscode ist für dieses Beispiel zusätzlich die Definition einer GUID für den Aktor, für den die Instruktion bestimmt ist, notwendig. Der Initialisierungs-Code verändert sich also entsprechend Listing 64.

```
01 #include <Orchid2.h>
02 #include <OrchidInstruction.h>
03 #include <Strings.h>
04
05 byte mac[] = { 0x12, 0x23, 0x34, 0x45, 0x56, 0x67 };
06 byte gateway[] = { 192, 168, 1, 1 };
07 byte subnet[] = { 255, 255, 255, 255 };
08 byte ip[] = { 192, 168, 1, 150 };
09 byte server[] = { 192, 168, 1, 140 };
10 char* client_id = "469abf7f-c733-4a2e-b469-7baa15d77673";
11 int port = 29100;
12 char* dev_key = "8178b0492fad1e60f59706d84146f17b4bf1afeaba9c2b7dd3db102450cb6b9c18347806";
13
14 char* actor_id = "49c77f2b-0bd9-49a5-a940-50ff8bb2d9c3";
15
16 Orchid orchid(mac, gateway, subnet, ip, port, server, client_id, dev_key);
17
18 void setup()
19 {
20     orchid.reset();
21 }
22
23 void loop()
24 {
25     //...
26 }
```

Listing 64 - Initialisierung zum Abrufen von Instruktionen

Die GUID des Aktors wird in Zeile 14 definiert und initialisiert.

Um eine Instruktion von einem ORCHID Server abzurufen, stellt die Client-Bibliothek die Methode *`getNextInstruction(char*)`* zur Verfügung. Dieser Methode muss die eindeutige ID des Aktors übergeben werden, für den nach einer auszuführenden Instruktion gesucht werden soll (diese ID muss dem ORCHID Server bekannt sein, da nur Instruktionen für bekannte Aktoren abgerufen werden können).

Die Methode liefert ein Objekt vom Typ *`OrchidInstruction`* zurück, das alle Daten enthält, die zur Ausführung einer Instruktion notwendig sind. Das folgende Listing 65 zeigt die Verwendung eines solchen Objekts.

```
23 void loop()
24 {
25     OrchidInstruction i = orchid.getNextInstruction(actor_id);
26
27     if(!i.isEmpty())
28     {
29         if(i.getAction() == Orchid::ACTION_DW)
30         {
31             if(strcmp(i.getValue(), "HIGH" == 0))
32             {
33                 digitalWrite(actor_pin, HIGH);
34             }
35             else if(strcmp(i.getValue(), "LOW" == 0))
36             {
37                 digitalWrite(actor_pin, LOW);
38             }
39             else { /* Do nothing */ }
40         }
41     }
42 }
```

Listing 65 - Abrufen einer Instruktion

Zunächst wird in Zeile 25 über die Methode *getNextInstruction(char*)* der Klasse *Orchid* eine Instanz der Klasse *OrchidInstruction* erzeugt. Diese Instanz ist entweder die nächste durchzuführende Instruktion für den angegebenen Aktor oder die leere Instruktion. Ob die Instruktion tatsächlich Anweisungen enthält, wird in Zeile 27 mit dem Aufruf der Methode *isEmpty()* der Instanz geprüft. Ist die Instruktion leer, wird die Verarbeitung nicht fortgeführt und die *loop()*-Methode beginnt von vorne.

Liegen hingegen Anweisungen in der Instruktion vor, wird zunächst die Aktion abgefragt, die ausgeführt werden soll. Dies geschieht über die Methode *getAction()* der Klasse *OrchidInstruction*. Diese Methode liefert einen numerischen Wert zurück, der für die Aktion steht, die der Aktor ausführen soll. Entsprechende Konstanten in der Klasse *Orchid* erleichtern die Verwendung, indem die Zahlenwerte durch aussagekräftige Namen ersetzt werden. Die folgende Tabelle 100 zeigt die verschiedenen Zahlenwerte sowie die zugehörigen Namen und erläutert die Bedeutung jedes einzelnen Werts.

Wert	Name	Bedeutung
1666	*ACTION_DW*	Es soll auf einen digitalen I/O-Pin geschrieben werden.
2666	*ACTION_DR*	Es soll von einem digitalen I/O-Pin gelesen werden.
3666	*ACTION_AW*	Es soll auf einen analogen I/O-Pin geschrieben werden.
4666	*ACTION_AR*	Es soll von einem analogen I/O-Pin gelesen werden.
5666	*ACTION_NONE*	Es wurde keine Aktion definiert.
6666	*ACTION_OTHER*	Es wurde eine unbekannte Aktion definiert.

Tabelle 100 - Action-Codes und entsprechende Konstanten

In Zeile 29 des obigen Beispiels wird die in der abgerufenen Instruktion enthaltene Aktion mit der Konstanten *ACTION_DW* verglichen, es wird also geprüft, ob ein Wert auf einen digitalen I/O-Pin geschrieben werden soll. Ist dies nicht der Fall, bricht die Verarbeitung ab, da für den Aktor, für den die Instruktion abgefragt wurde, keine anderen Operationen unterstützt werden.

Gleicht die Aktion der Konstanten `ACTION_DW`, wird in Zeile 31 überprüft, ob der auf den digitalen I/O-Pin zu schreibende Wert der Zeichenkette *„HIGH“* gleicht (Instruktionswerte werden stets als Zeichenkette übermittelt). Ist dies der Fall, wird an den Pin, an dem der Aktor, für den die Instruktion bestimmt ist, angeschlossen ist, der Wert `HIGH` geschrieben. Ist dies nicht der Fall und gleicht der zu schreibende Wert der Zeichenkette *„LOW“* (Zeile 35), wird der Wert `LOW` auf den Aktor-Pin geschrieben. Alle weiteren Werte sind ungültig (digitale I/O-Pins kennen nur die Werte `HIGH` und `LOW`) und werden ignoriert (Zeile 39).

Zusammengefasst ruft das hier gegebene Beispiel zunächst die nächste Instruktion für einen Aktor ab. Danach wird geprüft, ob sich Aktion und übermittelter Wert innerhalb der Instruktion (falls diese nicht leer ist) von dem betroffenen Aktor ausführen lassen und, wenn dies der Fall ist, die übermittelte Aktion mit dem übermittelten Wert ausgeführt.

Um ein Gerät über eine solche Instruktion einzuschalten, müsste der Entwickler also als Aktion *„`ACTION_DW`“* und als Wert *„`HIGH`“* übertragen. Der Sketch würde `HIGH` auf den Pin, an dem der Aktor angeschlossen ist, schreiben und der Stromkreis des Geräts geschlossen werden. Entsprechend würde die Aktion *„`ACTION_DW`“* mit dem Wert *„`LOW`“* das Gerät ausschalten.

8.4.5 Direktes Ausführen einer Instruktion

Das vorherige Beispiel hat gezeigt, dass es mit der Client Bibliothek möglich ist, eine Instruktion von einem ORCHID Server abzurufen, sie zu analysieren und entsprechende Schritte einzuleiten. Die dazu notwendige Menge an Code ist jedoch im Vergleich zum Nutzen sehr groß (16 Zeilen Code, um ein Gerät ein- oder auszuschalten).

Für die Aufgabe, einen Aktor ein- oder auszuschalten, stellt die Client-Bibliothek deshalb mit `executeNextInstruction(char*)` eine weitere Methode zur Verfügung, die es einem Entwickler erlaubt, eine Instruktion abzurufen und direkt ausführen zu lassen (mit der Einschränkung, dass dies nur für die Aktion `ACTION_DW` und die Werte `HIGH` und `LOW` möglich ist).

Das in Listing 65 gezeigte Beispiel verändert sich bei Verwendung der Methode `executeNextInstruction(char*)` entsprechend Listing 66.

```
23 void loop()
24 {
25     orchid.executeNextInstruction(actor_id);
25 }
```

Listing 66 - Direktes Ausführen einer Instruktion

8.4.6 Rückgabewerte und Fehlercodes verwenden

Die verschiedenen Überladungen der in 8.4.3 beschriebenen Methode `push()` liefern bei ihrer Ausführung einen Wahrheitswert (`bool`) zurück, der für den Erfolg oder Misserfolg der Methode steht. Die Methoden geben `true` zurück, wenn der Server auf ihre Anfragen mit dem HTTP-Code 200 (*„OK“*) antwortet, ansonsten wird `false` zurückgegeben. Die Methode `getNextInstruction(char*)` gibt ein Objekt vom Typ `OrchidInstruction` zurück, die Methode `heartbeat()` verfügt hingegen über keinen Rückgabetyp (`void`). Aus dem Rückgabewert einer Methode kann also nicht auf eine Fehlerursache oder die eigentliche Server-Antwort geschlossen werden.

Deshalb bietet die Klasse Orchid der Client-Bibliothek die Möglichkeit an, auf den letzten Fehler-Code zurückzugreifen. Jede Methode setzt diesen Code während ihrer Ausführung abhängig von ihrem Erfolg oder Misserfolg.

Der letzte Fehlercode kann über die Methode `getLastError()` der Klasse `Orchid` abgefragt werden und enthält zu jedem Zeitpunkt einen der in Tabelle 101 genannten Werte.

Code	Name	Bedeutung
0	*ORCHID_OK*	Kein Fehler.
-1	*ORCHID_NO_CONNECTION*	Es konnte keine Verbindung zum angegebenen Server hergestellt werden.
-2	*ORCHID_NO_ANSWER*	Der Server hat keine Antwort gesendet.
-3	*ORCHID_TIMEOUT*	Der Server hat die maximale Zeitspanne zwischen Anfrage und Antwort überschritten.
-4	*ORCHID_NO_DATA*	Es wurden keine Daten übermittelt.
-5	*ORCHID_SERVER_ERROR*	Ein interner Serverfehler trat auf.
-6	*ORCHID_UNKNOWN_VALUE*	Der Server hat Daten übermittelt, die nicht verwendet werden können.
-7	*ORCHID_BAD_REQUEST*	Der vom Client übermittelte Request war ungültig oder fehlerhaft.
-8	*ORCHID_UNAUTHORIZED*	Der Server hat die Anfrage abgewiesen, da der Client sich nicht ordnungsgemäß authentifizieren konnte.

Tabelle 101 - Fehlercodes

8.5 Problembehebung

Es kann vorkommen, dass sich ein Sketch nach der Einbindung der Client-Bibliothek nicht mehr kompilieren lässt. Beinhaltet die entsprechende Fehlermeldung den im folgenden Listing gezeigten Text, muss zusätzlich zur Client-Bibliothek auch die Ethernet-Bibliothek eingebunden werden.

```
error: Ethernet.h: No such file or directory
```

Listing 67 - Fehlermeldung der Arduino IDE

Tritt dieser Fehler auf, muss die in Listing 68 gezeigte Zeile am Anfang des Sketches hinzugefügt werden.

```
01 #include <Ethernet.h>
```

Listing 68 - Fehlerbehebung durch Einbindung von Ethernet.h

8.6 Veränderung und Erweiterung der Client-Bibliothek

Die Veränderung und Erweiterung der Client-Bibliothek wird im Folgenden exemplarisch beschrieben. Dazu wird zunächst, als Veränderung, den Methoden, mit denen Sensordaten auf einen ORCHID Server geladen werden, ein neues Header-Feld hinzugefügt. Im Anschluss wird der Klasse Orchid eine neue Methode hinzugefügt, die in der Lage ist, HEAD-Requests an einen ORCHID Server zu übermitteln (in 7.7.2 wurde entsprechend beschrieben, wie die Server-Komponente um die Behandlung solcher Requests erweitert werden kann).

8.6.1 Veränderung

Wie bereits in Abschnitt 6.6.1 beschrieben, setzt das ORCHID Toolkit keine Verschlüsselung der Daten, die zwischen Server und Client übertragen werden, ein. Um verschiedene kryptografische Verfahren nutzen zu können, müssen Server und Client in der Lage sein, den in einer Übertragung verwendeten Verschlüsselungsalgorithmus zu erkennen. Über ein Header-Feld kann diese Information komfortabel übermittelt werden.

Für das folgende Beispiel wird angenommen, dass das hinzuzufügende Header-Feld dem in Listing 69 gezeigten entspricht.

```
X-Content-Encryption: XTEA
```

Listing 69 - Beispielhaftes Header-Feld zur Inhaltsverschlüsselung

Würde dieser Header exakt wie im obigen Listing angegeben übertragen, könnte ein entsprechend angepasster ORCHID Server daraus schließen, dass zur Datenverschlüsselung der *XTEA*-Algorithmus zum Einsatz kommen soll. *XTEA* (e**X**tended **T**iny **E**ncryption **A**lgorithm) ist eine Erweiterung des für Angriffe anfälligen *TEA* (**T**iny **E**ncryption **A**lgorithm), die aufgrund ihrer geringen Größe für die Implementierung in einem ORCHID Client geeignet erscheint (vgl. [26], S. 148-149).

Die Client-Bibliothek stellt zahlreiche Überladungen der Methode `push()` zur Verfügung, mit denen Daten vieler verschiedener Datentypen übertragen werden können. Um ein neues Header-Feld hinzuzufügen, muss jedoch nur die zentrale Methode `push_prefix(char*)` verändert werden, die in der Datei `Orchid.h` definiert ist.

Diese Methode nutzt eine Instanz der Klasse `Client`, die in der Ethernet-Library enthalten ist, um einen ausgehenden Request zu erzeugen. Eine solche Instanz verhält sich ähnlich wie eine Datei und bietet die Methoden `print()` und `println()` an, die Daten an einen Request anhängen.

Das Protokoll, das in 5.8 ausführlich beschrieben wurde, gibt vor, dass Header-Felder durch die Zeichenfolge *„Carriage Return Line Feed“* (*CRLF*, auch `\r\n`) voneinander getrennt werden müssen. Die Methode `println()` der Klasse `Client` fügt diese Zeichenfolge automatisch an.

Das folgende Listing 70 zeigt einen Ausschnitt der Methode `push_prefix(char*)`. Diesem Ausschnitt wurde das neue Header-Feld bereits hinzugefügt (fett hervorgehoben).

```
01 if(client.connect())
02 {
03     client.println("PUT HTTP/1.1");
04     client.print("Board-ID: ");
05     client.println(_board_id);
06     client.println("User-Agent: Orchid Arduino Toolkit");
07     // ...
08     client.println("X-Content-Encryption: XTEA");
09     //...
10     client.print("Authorization: ");
11     client.println(_devkey);
12     //...
13 }
```

Listing 70 - Hinzufügen des neuen Header-Felds (1)

Würde der Name des verwendeten kryptografischen Algorithmus in einer Instanzvariablen (wie im obigen Listing der Developer-Key, der im Header-Feld *Authorization* übertragen wird) namens „*_cryptoName*“ verwaltet, würde sich der Code entsprechend dem folgenden Listing 71 verändern.

```
01 if(client.connect())
02 {
03     client.println("PUT HTTP/1.1");
04     client.print("Board-ID: ");
05     client.println(_board_id);
06     client.println("User-Agent: Orchid Arduino Toolkit");
07     // ...
08     client.print("X-Content-Encryption: ");
09     client.println(_cryptoName);
10     //...
11     client.print("Authorization: ");
12     client.println(_devkey);
13     //...
14 }
```

Listing 71 - Hinzufügen des neuen Header-Felds (2)

Nach diesen einfachen Änderungen überträgt die Client-Bibliothek in jedem PUT-Request das neue Header-Feld.

8.6.2 Erweiterung

In Abschnitt 7.7.2 wurde beschrieben, wie die Server-Komponente so erweitert werden kann, dass HEAD-Requests eines Clients von ihr verarbeitet und beantwortet werden können. Entsprechend diesem Beispiel soll nun der Client-Bibliothek eine Methode hinzugefügt werden, die HEAD-Requests an einen ORCHID Server sendet.

Es steht einem Entwickler frei, die neue Methode zunächst in der Datei *Orchid.h* zu deklarieren und ihre eigentliche Implementierung in der Datei *Orchid.cpp* vorzunehmen, oder aber die gesamte Methode direkt in *Orchid.h* zu implementieren. Der Einfachheit halber nutzt dieses Beispiel die zweite Möglichkeit.

Um einen Request an einen Server zu senden, muss zunächst eine Instanz der Klasse *Client* aus der Ethernet-Library erzeugt werden. Über diese Instanz werden nun die benötigten Header-Felder des Requests an den Server übermittelt (ggf. zusätzlich ein Message-Body). Da ein Server nicht immer ohne Zeitverzögerung eine Antwort senden kann, wird nach dem Senden der Anfrage eine bestimmte Zeit gewartet. Hat der Server auch nach dieser Zeit keine Antwort gesendet, ist ein Fehler aufgetreten, der Server ist nicht erreichbar oder antwortet nicht. Treffen Daten ein, können sie Byte für Byte aus der Antwort ausgelesen und interpretiert werden. Das folgende Beispiel zeigt die Interpretation aufgrund ihres Umfangs nicht, sondern beschreibt lediglich, welche Aktionen durchgeführt werden müssen. Ist die Antwort vollständig ausgelesen und die darin enthaltenen Werte interpretiert worden, wird die Client-Instanz über die Methode *stop()* angehalten und der ermittelte Wert zurückgegeben.

Das nachfolgende Listing 72 zeigt die neue Methode, über die die Laufzeit eines ORCHID Servers in einem Sketch abgefragt werden kann.

```
long requestServerRuntime()
{
    _last_error = Orchid::ORCHID_OK;
    long ret = -1;

    Client client(_server, remotePort);

    if(client.connect())
    {
        client.println("HEAD HTTP/1.1");
        client.println("User-Agent: Orchid Arduino Toolkit");
        client.print("Board-ID: ");
        client.println(_board_id);
        client.print("Authorization: ");
        client.println(_devkey);
        client.println("\r\n");
    }
    else
    {
        _last_error = Orchid::ORCHID_NO_CONNECTION;
    }

    waitForClient(client);

    if(client.available())
    {
        //Read answer, incl. status code and message
        //Store message value as long in variable "ret"
    }
    else
    {
        _last_error = Orchid::ORCHID_NO_ANSWER;
    }

    if(!client.connected())
    {
        client.stop();
    }

    return ret;
}
```

Listing 72 - Erweiterung der Client-Komponente

9 Handbuch Zusatztools

Dieser Abschnitt beschreibt die im ORCHID Toolkit enthaltenen Zusatztools *„ggen“* und *„dkgen“*. Die notwendigen Systemvoraussetzungen für den Betrieb dieser Tools entsprechen denen der Server-Komponente (vgl. 7.1) und werden hier nicht gesondert erwähnt.

9.1 ggen

Das Tool *„ggen“ („**GUID Gen**erator“)* dient zur Erzeugung von GUIDs (vgl. 6.3.1) für Clients, Sensoren oder Aktoren. Es handelt sich um ein Kommandozeilenprogramm, auf eine grafische Benutzeroberfläche wurde bewusst verzichtet.

Um mit *ggen* eine GUID zu erzeugen, reicht es, das Programm über die Kommandozeile aufzurufen, wie in der folgende Abbildung zu sehen ist.

```
steps@zombie:~/Masterarbeit/dvd/bin/mono/tools$ ./ggen
c6193a8b-502e-4652-86d7-baae4bade42f
steps@zombie:~/Masterarbeit/dvd/bin/mono/tools$ _
```

Abbildung 44 - Ausgabe von ggen

Bei jedem Aufruf des Tools wird eine neue GUID erzeugt. Sollen mehrere GUIDs auf einmal erzeugt werden, kann dies über den Parameter `-c` bzw. `--count` erreicht werden. Dieser Parameter erwartet die Angabe der Anzahl der zu erzeugenden GUIDs, wie Abbildung 45 zeigt.

```
steps@zombie:~/Masterarbeit/dvd/bin/mono/tools$ ./ggen -c 5
bca904e5-8658-41a6-85c5-95bb41efdd67
c87f7c56-236a-4a97-b1ca-b79eb372f8c1
dedaad76-70a3-4bac-9ec2-f21c741ede8f
7933f2ba-5f66-421e-9990-294ff5cbc69f
a15a1926-e84c-4229-bae6-3649acde9074
steps@zombie:~/Masterarbeit/dvd/bin/mono/tools$ _
```

Abbildung 45 - Erzeugung mehrerer GUIDs mit ggen

Jede mit *ggen* erzeugte GUID ist eindeutig, d.h. das Tool erzeugt niemals zweimal die gleiche ID (im Rahmen der Möglichkeiten von GUIDs, siehe 6.3.1 für Details). Eine auf diese Weise erzeugte ID kann an allen Stellen des ORCHID Toolkits eingesetzt werden, an denen eine eindeutige Kennung gefordert wird.

9.2 dkgen

Die Server-Komponente des Toolkits ist in der Lage, die Autorisierungsinformationen von Clients oder Drittsoftware (den sog. *„Developer Key“*, vgl. 6.3.3.2) zu prüfen. Obwohl eine beliebige Zeichenkette als Developer Key verwendet werden kann, wird empfohlen, nur Schlüssel zu nutzen, die mit dem Tool *dkgen (**D**eveloper **K**ey **Gen**erator)* erzeugt wurden.

Jeder Aufruf des Tools erzeugt einen neuen, zufälligen Developer Key (zwei auf diese Weise erzeugte Developer Keys sind niemals[69] gleich), wie die folgende Abbildung zeigt.

```
steps@zombie:~/Masterarbeit/dvd/bin/mono/tools$ ./dkgen
607bf6aaa0197bac213ecc556bbd096744edaad62cc9d20963a692774d8adef165efa677
steps@zombie:~/Masterarbeit/dvd/bin/mono/tools$ _
```

Abbildung 46 - Erzeugung eines Developer Keys mit dkgen

[69] Gemeint ist hier „mit ausreichend hoher Wahrscheinlichkeit“, vgl. 6.3.1.

10 Endbenutzerhandbuch

Nachdem in den Abschnitten 7, 8 und 9 die Verwendung und Erweiterung der Toolkit-Komponenten und Zusatztools beschrieben wurde, befasst sich dieser Abschnitt mit der letztendlichen Verwendung des ORCHID Toolkits für eigene Projekte.

10.1 Einleitung

Auf dem dieser Arbeit beiliegenden Datenträger befindet sich neben allen zu Nutzung des Toolkits notwendigen Dateien auch ein Abbild einer virtuellen Maschine (*VM*) für die Virtualisierungssoftware *Oracle VM VirtualBox*[70].

VirtualBox kann kostenlos für Windows, Linux und MacOS von der Herstellerwebsite[71] bezogen werden und ist in der Lage, einen vollwertigen Computer zu virtualisieren, so dass nahezu beliebige Betriebssysteme in einer virtuellen Umgebung ausgeführt werden können.

Auf der VM wurden im Vorfeld ein Linux-System (Linux Mint[72]) und alle weiteren Komponenten installiert, die zum Betrieb eines ORCHID Servers und zur Entwicklung von Sketches mit der Client-Bibliothek notwendig sind.

10.2 Einrichtung der virtuellen Maschine

Alle zum Import der virtuellen Maschine in VirtualBox notwendigen Dateien befinden sich auf der beigelegten DVD im Verzeichnis `VM`.

Nach dem Start von VirtualBox muss zunächst über den Button *„Neu“* eine neue virtuelle Maschine hinzugefügt werden. Der daraufhin erscheinende Assistent ist selbsterklärend, verlangt aber einige spezifische Einstellungen, die der folgenden Tabelle entnommen werden können.

Einstellung	Wert
Name	Beliebig wählbar.
Typ des Gastbetriebssystems	Linux (Version: Ubuntu)
Größe Hauptspeicher	Mindestens 512 MB; empfohlen: 1024 MB oder mehr.

Tabelle 102 - Einstellungen der virtuellen Maschine

Im letzten Schritt des Assistenten wird nach den Einstellungen für eine virtuelle Festplatte gefragt. Ein entsprechendes Festplattenabbild liegt der DVD unter dem Dateinamen `OrchidVM.vdi` im Verzeichnis `VM` der DVD bei. Über den Button am rechten Rand des Fensters kann diese Datei als Festplatte ausgewählt werden.

Die folgende Abbildung zeigt die notwendigen Einstellungen für die virtuelle Festplatte.

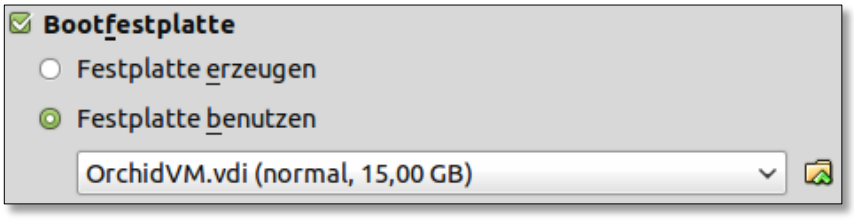

Abbildung 47 - Auswahl einer virtuellen Festplatte

[70] http://www.virtualbox.org/
[71] http://www.virtualbox.org/wiki/Downloads
[72] http://www.linuxmint.com/

Nach diesem Schritt ist die Einrichtung der virtuellen Maschine abgeschlossen und sie kann verwendet werden. Im Hauptfenster von VirtualBox erscheint sie auf der linken Seite und kann über einen Doppelklick gestartet werden.

10.3 Übersicht

Nach dem Start der virtuellen Maschine (Benutzername und Passwort lauten `orchid`) wird der Desktop angezeigt, der Abbildung 48 ähnelt.

Abbildung 48 - Desktop der virtuellen Maschine

Auf dem Desktop befinden sich Verknüpfungen zu den wichtigsten für den Betrieb eines ORCHID Servers notwendigen Dateien und Programmen.

Datei/Verknüpfung	Beschreibung
`Computer`	Öffnet eine Ansicht des Dateisystems.
`orchid's Home`	Öffnet das Heimverzeichnis des Benutzers *orchid*.
`Arduino IDE`	Startet die Arduino Entwicklungsumgebung.
`Firefox Web Browser`	Startet den Firefox-Webbrowser[73].
`MonoDevelop`	Startet die Entwicklungsumgebung MonoDevelop.
`Orchid_Server.sln`	Öffnet das Projekt der Server-Komponente mit MonoDevelop.
`stop_orchid.sh`	Beendet einen laufenden ORCHID Server auf der virtuellen Maschine.
`restart_orchid.sh`	Startet einen laufenden ORCHID Server neu.
`start_orchid.sh`	Startet einen ORCHID Server auf der virtuellen Maschine.
`Orchid Arduino Toolkit Live Demo`	Startet den Firefox-Webbrowser und navigiert zur Live-Demonstration des ORCHID Toolkit.
`Terminal`	Öffnet die Kommandozeile.

Tabelle 103 - Desktop-Elemente

[73] http://www.mozilla.org/de/firefox/

Die folgende Software ist auf der virtuellen Maschine bereits installiert:

Software	Version
Mono	2.6.7
Java SE Laufzeitumgebung	1.6.0_24
Arduino IDE	0022
MonoDevelop	2.4
XAMPP for Linux	1.7.4
MySQL Server	5.5.8

Tabelle 104 - Auf der VM installierte Software

10.4 Einrichtung der MySQL-Datenbank

Die MySQL-Datenbank, die für den Betrieb eines ORCHID Servers benötigt wird, kann über das zum Lieferumfang des XAMPP Server-Pakets gehörende Programm *phpMyAdmin*[74] komfortabel eingerichtet werden[75]. Zunächst muss dazu der Apache Webserver über die Kommandozeile gestartet werden. Den entsprechenden Befehl zeigt Abbildung 49.

```
orchid@orchidvm ~ $ sudo /opt/lampp/lampp start
[sudo] password for orchid:
Starting XAMPP for Linux 1.7.4...
XAMPP: Starting Apache with SSL (and PHP5)...
XAMPP: Starting MySQL...
XAMPP: Starting ProFTPD...
XAMPP for Linux started.
orchid@orchidvm ~ $
```

Abbildung 49 - Starten von XAMPP (Linux)

Auf der virtuellen Maschine läuft nun sowohl ein Apache Webserver, als auch eine MySQL Server-Instanz.

phpMyAdmin ist eine Anwendung, die im Browser aufgerufen wird. Sie ist unter der Adresse `http://localhost/phpmyadmin` zu finden und als Startseite des installierten Firefox-Webbrowsers eingetragen. Beim ersten Aufruf dieser Adresse verlangt XAMPP for Linux nach einem Benutzernamen und einem Passwort. Diese lauten `lampp` und `orchid`.

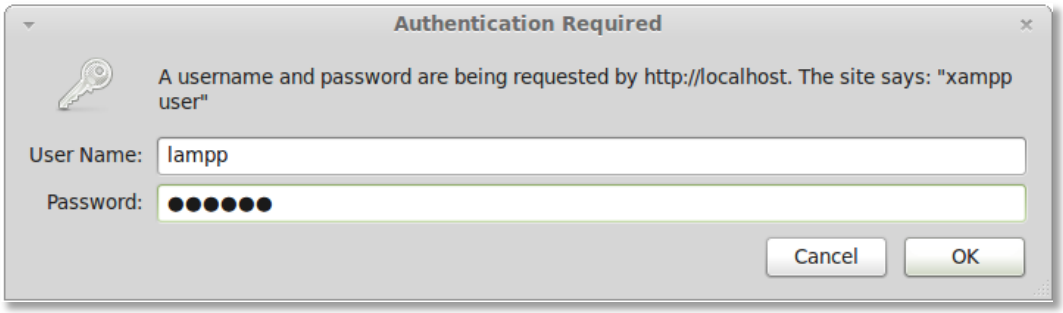

Abbildung 50 - Anmeldung am lokalen Webserver

[74] http://www.phpmyadmin.net
[75] Auf der beiliegenden virtuellen Maschine ist die Datenbank bereits eingerichtet.

Da phpMyAdmin auf den lokalen MySQL Server zugreift, benötigt es den Benutzernamen und das Passwort des Datenbankadministrators. Der Benutzername des Administrators lautet `root`, das Passwort lautet `orchid`.

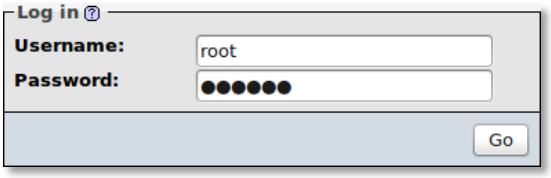

Abbildung 51 - Login in phpMyAdmin

Nach der Bestätigung von Benutzername und Passwort wird das Hauptfenster von phpMyAdmin angezeigt. Im Zentrum des Fensters kann eine neue Datenbank angelegt werden, wie in der folgenden Abbildung gezeigt wird.

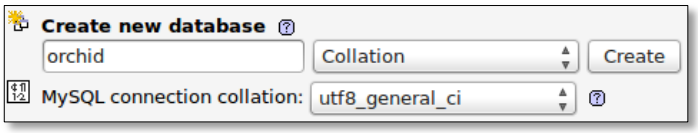

Abbildung 52 - Dialog zur Erzeugung einer neuen Datenbank

Nachdem die neue Datenbank erzeugt wurde, wird sie automatisch als aktive Datenbank ausgewählt. Im oberen Bereich des dargestellten Fensters befindet sich der Reiter *„Import“*, über den der Dialog zum Import einer bestehenden Datenbankstruktur aufgerufen wird. Wie in Abbildung 53 dargestellt, kann über diesen Dialog eine Datei für den Import ausgewählt werden.

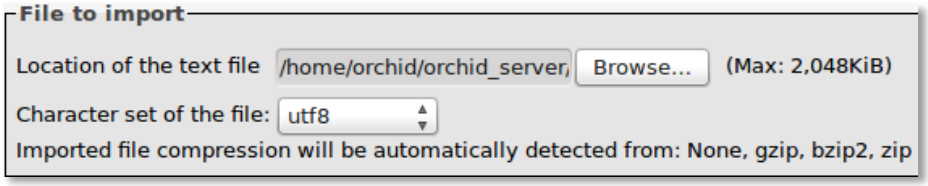

Abbildung 53 - Dialog zum Import einer SQL-Datei

Die Datenbankstruktur befindet sich in der Datei `/home/orchid/orchid_server/orchid.sql`.

Nach der Bestätigung des Dateiimports über den Button *„Go“* wird die angegebene Datei importiert und eine Erfolgsmeldung ausgegeben. Die Datenbank ist nach diesen Schritten einsatzbereit.

10.5 Hinzufügen eines ORCHID Clients

Ein ORCHID Server nimmt nur Anfragen von Clients entgegen, die sich über eine GUID identifizieren, die dem Server bekannt ist. Entsprechend muss, wenn ein neuer Client hinzugefügt werden soll, diesem eine solche ID zugewiesen und in die Datenbank eingetragen werden.

Zur Erzeugung einer neuen GUID kann das Zusatztool `ggen` oder das Programm `uuid` verwendet werden. Auf der virtuellen Maschine ist uuid bereits installiert und kann über die Kommandozeile aufgerufen werden. Der Aufruf entspricht dem in Abbildung 54 gezeigten.

```
orchid@orchidvm ~ $ uuid
12098346-da45-11e0-a33a-080027c251f7
orchid@orchidvm ~ $
```

Abbildung 54 - Ausgabe von uuid

Ein neuer Client kann entweder über phpMyAdmin oder über die Kommandozeile hinzugefügt werden.

10.5.1 Hinzufügen mit phpMyAdmin

Um einen Client unter Verwendung von phpMyAdmin hinzuzufügen, muss die Anwendung im Browser aufgerufen (`http://localhost/phpmyadmin`) und die Datenbank `orchid` ausgewählt werden.

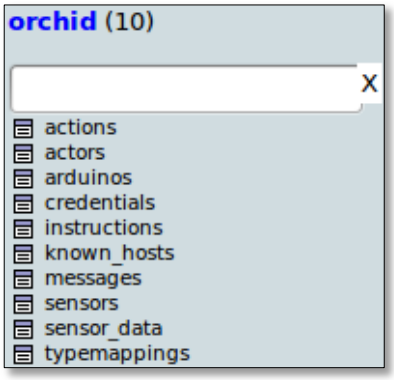

Abbildung 55 - Auflistung von Datenbanktabellen in phpMyAdmin

Wie in der obigen Abbildung zu sehen, werden daraufhin alle in der Datenbank vorhandenen Tabellen angezeigt. Um einer Tabelle einen Datensatz hinzuzufügen, muss diese in der abgebildeten Auswahlliste angeklickt werden, woraufhin alle bereits enthaltenen Datensätze und das Menü zum Bearbeiten der Tabelle dargestellt werden. Über die Auswahl des Menüpunkts *„Insert“* wird die Eingabemaske für neue Datensätze aufgerufen, die in Abbildung 56 zu sehen ist.

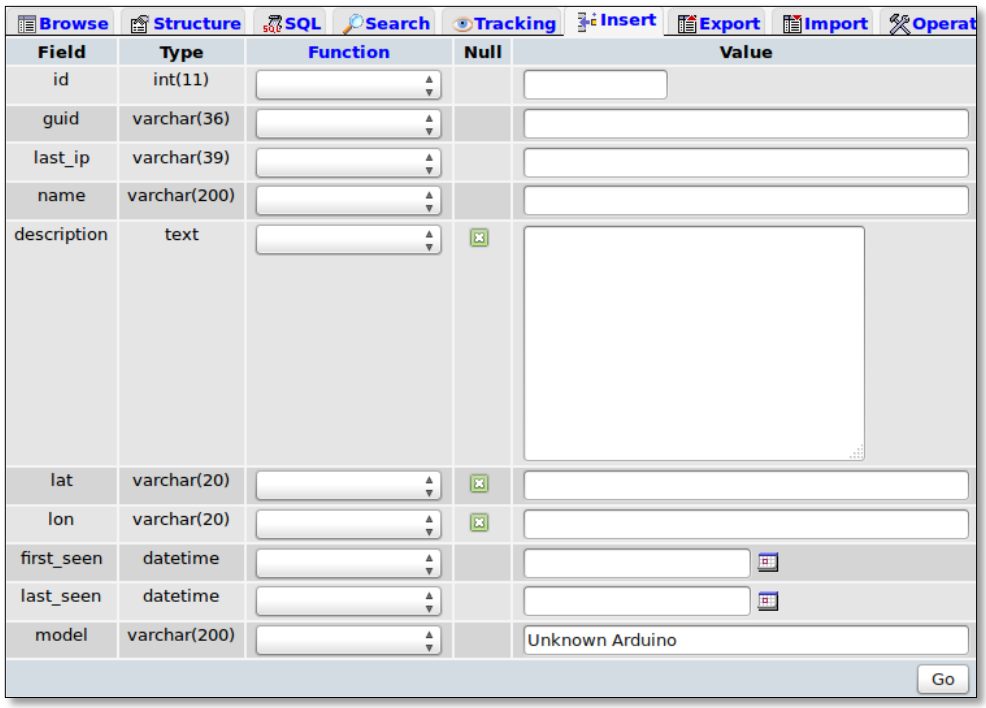

Abbildung 56 - Eingabemaske für neue Datensätze

Die Bedeutung der auszufüllenden Felder kann Abschnitt 5.10 entnommen werden. Als Wert für das Feld `last_ip` kann eine frei wählbare IP-Adresse eingegeben werden, da diese bei der ersten Verbindung des neuen Clients mit dessen aktueller IP-Adresse überschrieben wird.

10.5.2 Hinzufügen über die Kommandozeile

Um einen neuen Client über die Kommandozeile hinzuzufügen, kann das Shellscript `/home/orchid/scripts/add_client.sh` oder das Kommandozeilenwerkzeugt *mysql* verwendet werden.

Die folgende Abbildung zeigt die Verwendung des Scripts mit optionalen Parametern. Wird es ohne Parameter aufgerufen, werden alle Felder mit Standardwerten gefüllt.

```
orchid@orchidvm ~/scripts $ ./add_client.sh -n "Neuer Client" -i "127.0.0.1" -d
"Beschreibung"
Starting XAMPP for Linux 1.7.4...
XAMPP: XAMPP-Apache is already running.
XAMPP: XAMPP-MySQL is already running.
XAMPP: XAMPP-ProFTPD is already running.
XAMPP for Linux started.
Client added:
  GUID: a435ffee-da48-11e0-ad93-080027c251f7
  Name:Neuer Client
  Description: Beschreibung
  Lat/Lon: 0/0
  First seen: 2011-09-08 20:30:34
  Last seen: 2011-09-08 20:30:34
  Model: Unknown Arduino
orchid@orchidvm ~/scripts $
```

Abbildung 57 - Hinzufügen eines neuen ORCHID Clients (Shellscript)

Die gezeigte GUID `a435ffee-da48-11e0-ad93-080027c251f7` wird im weiteren Verlauf dieses Handbuchs als ID verwendet und muss entsprechend angepasst werden.

10.6 Konfiguration und Start des ORCHID Servers

Das Verzeichnis `/home/orchid/orchid_server/bin/Settings` enthält die Konfigurationsdatei `Orchid.ini`, über die der ORCHID Server konfiguriert werden kann. Die enthaltenen Einstellungen entsprechen den in 7.5 beschriebenen und sind auf die Verwendung mit der virtuellen Maschine ausgelegt.

Der ORCHID Server kann entweder direkt über die Kommandozeile oder über das Shellscript `start_orchid.sh` auf dem Desktop gestartet werden. Der Start über das Shellscript wird empfohlen, da dieses bei Bedarf auch den MySQL-Server startet und der Referenzimplementierung des ORCHID Servers den Pfad zur Konfigurationsdatei mitteilt.

```
Starting LAMPP...
[sudo] password for orchid:
Starting XAMPP for Linux 1.7.4...
XAMPP: XAMPP-Apache is already running.
XAMPP: XAMPP-MySQL is already running.
XAMPP: XAMPP-ProFTPD is already running.
XAMPP for Linux started.
Starting Orchid Reference Server
Welcome to the Orchid Development Server
----------------------------------------

            _ (`-`) _
          /`  '.\ /.'  `\
          ``'-.,=,.-'``
             .'//\\'.
           (_/\ " /\_)
               '-'
    Orchid Server, V. 1.0.4249


(08.09.2011 19.45.00.252) [Info] Database connection established (root@localhost/orchid)
(08.09.2011 19.45.00.288) [Info] Database connection established (root@localhost/orchid)
(08.09.2011 19.45.00.308) [Info] Webserver is running (192.168.1.117:8181)
(08.09.2011 19.45.00.309) [Info] Server is running (192.168.1.117:29100)
```

Abbildung 58 - Ausgabe des Referenzservers

Abbildung 58 zeigt den erfolgreichen Start eines ORCHID Servers, der auf den Ports 29100 und 8181 auf eingehende Verbindungen wartet. Zum Start des MySQL-Servers sind Administratorrechte notwendig, weshalb das Script zur Passworteingabe auffordert.

10.7 Einbindung eines ORCHID Clients

Ist der ORCHID Server einsatzbereit und ist ihm die GUID des neuen Clients bekannt, kann dieser eingebunden, d.h. mittels der Client-Bibliothek mit dem Server verbunden werden. Zunächst muss dazu eine Schaltung aufgebaut werden, die Komponenten (Sensoren und/oder Aktoren) enthält, die in Verbindung mit dem ORCHID Toolkit verwendet werden sollen. Für das im Folgenden erarbeitete Beispiel wird angenommen, dass diese Schaltung der in Abbildung 59 gezeigten entspricht.

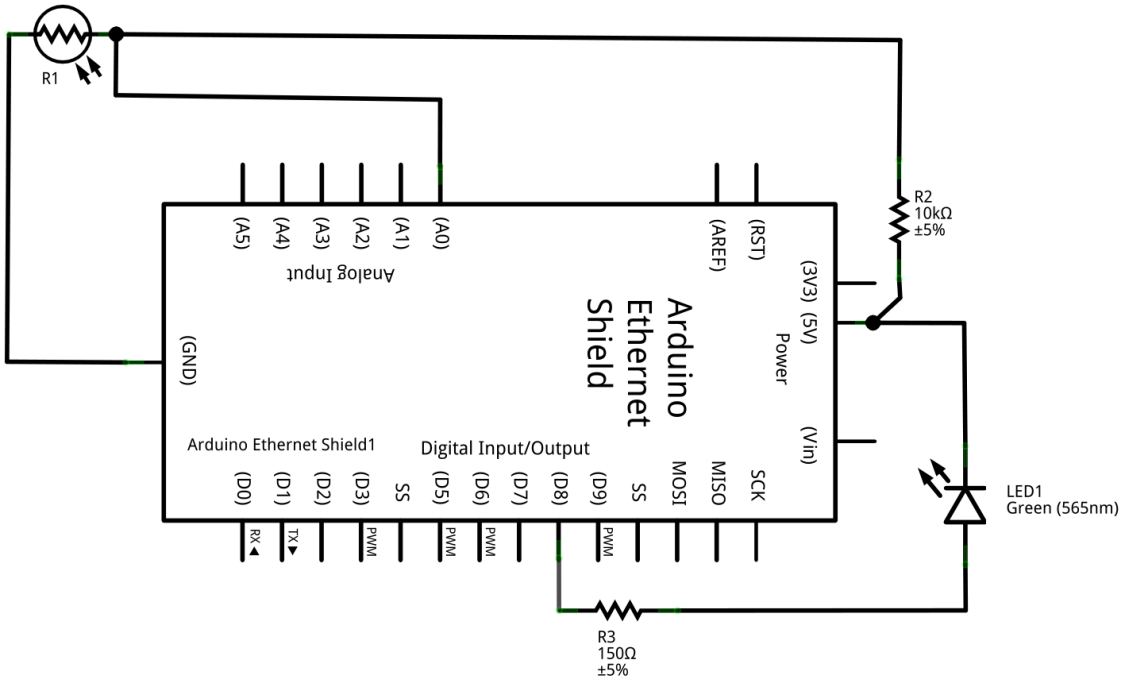

Abbildung 59 - Schaltplan der Beispielschaltung

Die obige Abbildung zeigt einen Photowiderstand (Lichtsensor) *R1*, der an Analog Input 0 des Arduino I/O-Boards angeschlossen ist. Eine grüne LED (*LED1)* ist mit Digital Input/Output 8 des Boards verbunden. Die Werte der Widerstände *R2* und *R3* variieren je nach verwendetem Bauteil.

Der Lichtsensor wird genutzt, um den Status einer Lampe zu beobachten. Ist die Lampe eingeschaltet, fällt also viel Licht auf den Sensor, verringert sich dessen Widerstand, die gemessenen Sensorwerte tendieren gegen 0. Ist die Lampe hingegen ausgeschaltet und fällt wenig Licht auf den Sensor, erhöht sich der Widerstand und die Sensorwerte tendieren gegen 1023[76]. Im Rückschluss kann bei einem sehr niedrigen Sensorwert (< 100) angenommen werden, dass die Lampe eingeschaltet und ansonsten, dass sie ausgeschaltet ist. Die LED dient als Repräsentation beliebiger, an einen Arduino Microcontroller anschließbarer Hardware.

Genau wie für einen Client selbst, müssen auch für an diesen Client angeschlossene Bauteile, die von einem ORCHID Server verarbeitet werden sollen, über eine GUID identifiziert werden. Tabelle 105 zeigt die in diesem Beispiel verwendeten IDs.

[76] Der Arduino rechnet gemessene Spannungen in Werte zwischen 0 und 1023 um.

Bauteil	GUID
Arduino	*a435ffee-da48-11e0-ad93-080027c251f7*
R1	*ba222420-da54-11e0-8879-080027c251f7*
LED1	*ef1c2504-da54-11e0-b860-080027c251f7*

Tabelle 105 - Beispiel-GUIDs

Der Photowiderstand und die LED sind dem ORCHID Server bisher nicht bekannt, es ist entsprechend nicht möglich, sie zu verwenden. Um sie dem Server bekannt zu machen, müssen sie in die Datenbanktabelle `actors` (*LED1*) bzw. `sensors` (*R1*) eingetragen werden. Es ist dabei sehr wichtig, dass im Feld `arduino_id` die numerische ID des neu hinzugefügten Clients ausgewählt wird, wie die folgende Abbildung zeigt.

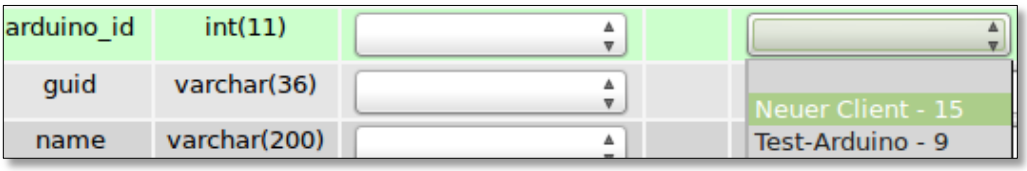

Abbildung 60 - Verweis auf einen Client

Die Eintragung verläuft analog zu der in 10.5.1 beschriebenen.

Nachdem die Schaltung aufgebaut und sowohl der Client, als auch die an diesen angeschlossenen Komponenten dem Server bekannt gemacht wurden, kann ein Sketch entwickelt werden, der die Funktionen der Client-Bibliothek nutzt. Da die Entwicklung von Sketches mit der Client-Bibliothek bereits in 8.4 ausführlich beschrieben wurde, wird an dieser Stelle nur der Code des Beispiel-Sketchs gezeigt, ohne im Detail darauf einzugehen. Kommentare wurden aus dem nachstehenden Listing 73 entfernt; eine kommentierte Version befindet sich auf dem dieser Arbeit beiliegenden Datenträger (vgl. Abschnitt 17).

```
#include <Ethernet.h>
#include <Strings.h>
#include <OrchidInstruction.h>
#include <Orchid2.h>

byte mac[] = { 0x00, 0x35, 0x92, 0x1D, 0x66, 0x35 };
byte gateway[] = { 192, 168, 1, 1 };
byte subnet[] = { 255, 255, 255, 0 };
byte ip[] = { 192, 168, 1, 146 };
byte server[] = { 192, 168, 1, 117 };

int port = 29100;

char* arduino_id = "a435ffee-da48-11e0-ad93-080027c251f7";
char* sensor_id = "ba222420-da54-11e0-8879-080027c251f7";
char* led_id="ef1c2504-da54-11e0-b860-080027c251f7";
char* devkey = "a2da24f8b064af8fdba7e31da4e2518f4b5785e307955723adf5da9fa604838f187ca324";

Orchid orchid(mac, gateway, subnet, ip, port, server, arduino_id, devkey);

void setup()
{
    Serial.begin(9600);
    delay(1500);
    pinMode(8, OUTPUT);
    orchid.reset();
}

void loop()
{
    boolean success = orchid.push(sensor_id, analogRead(0));

    Serial.print("Push: ");
    Serial.println(orchid.getLastError());

    int result = orchid.executeNextInstruction(led_id);

    Serial.print("Execute: ");
    Serial.println(orchid.getLastError());

    delay(2000);
}
```

Listing 73 - Beispielsketch

Die im obigen Listing gezeigten GUIDs und der Developer Key müssen dem verwendeten Server angepasst werden, damit der Sketch verwendet werden kann.

Wird dieser Sketch auf einem Arduino ausgeführt und ist der ORCHID Server für den Client erreichbar, erscheint auf dem seriellen Monitor der Arduino IDE eine Ausgabe, die der in Abbildung 61 entspricht.

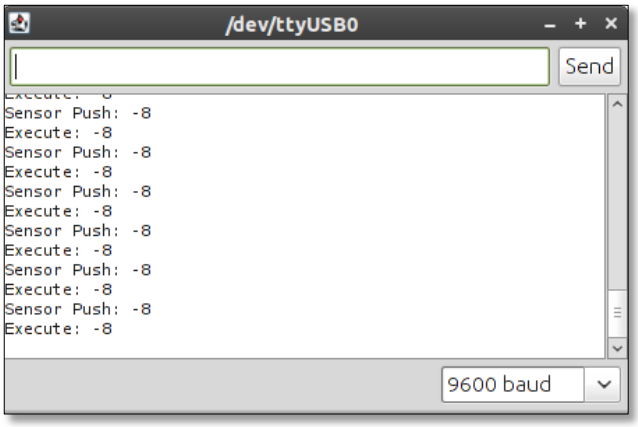

Abbildung 61 - Fehlercodes im seriellen Monitor der Arduino IDE

Beide ausgeführten Operationen (Hochladen von Sensordaten und direktes Ausführen der nächsten vorliegenden Instruktion) werden mit dem Fehlercode `-8` vom Server zurückgewiesen. Es handelt sich dabei nicht um einen Fehler, sondern um das erwartete Verhalten.

Der Server verfügt bisher noch nicht über einen Datenbankeintrag für die Berechtigungen des neuen Clients. Entsprechend werden alle Anfragen, die dieser an den Server stellt, mit HTTP-Code 401 („*Unauthorized*“) beantwortet.

Um dem neuen Client alle Operationen zu erlauben, muss ein Eintrag in die Datenbanktabelle `credentials` vorgenommen werden, wie Abbildung 62 zeigt.

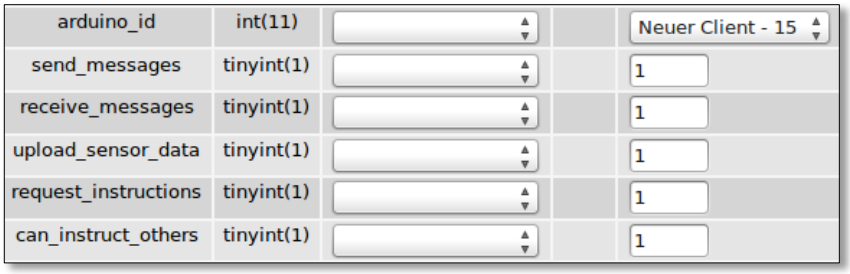

Abbildung 62 - Einfügen von Credentials in phpMyAdmin

Ist dieser Eintrag gespeichert, ist es dem neuen Client erlaubt, jede Operation auf dem Server durchzuführen (um eine Operation zu verbieten, muss als Wert `0` eingetragen werden). Die Ausgabe im seriellen Monitor verändert sich entsprechend (ohne Server-Neustart):

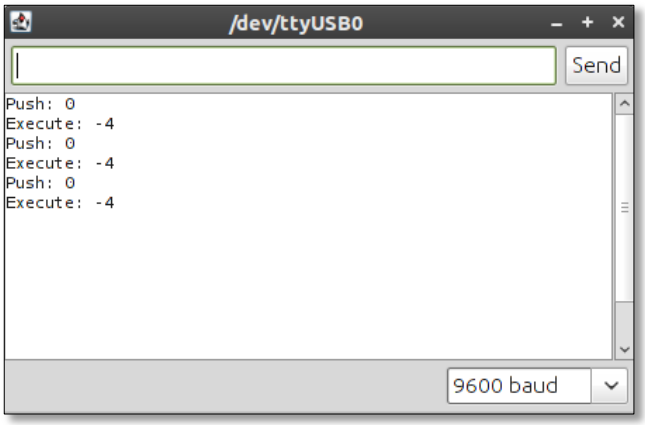

Abbildung 63 - Ausgabe im seriellen Monitor der Arduino IDE

Das Hochladen von Sensordaten ist nun möglich die Ausführung der nächsten Instruktion wird erwartungsgemäß mit Fehlercode -4 abgebrochen, da auf dem Server bisher keine Instruktionen für den neuen Client vorliegen.

10.8 Drittsoftware einbinden

Instruktionen werden von Drittsoftware an einen ORCHID Server übermittelt, die nicht Bestandteil dieser Arbeit ist. Als Proof of Concept befindet sich aber eine in PHP implementierte Bibliothek und eine entsprechende Demonstrationssoftware auf der virtuellen Maschine.

Die Software sendet HTTP-Anfragen (entsprechend den in 5.8 genannten Vorgaben) an einen ORCHID Server und wertet dessen Antworten aus. Die ermittelten Daten werden daraufhin aufbereitet und dargestellt.

Über die Adresse `http://localhost/orchid` wird die exemplarische Drittsoftware im Browser aufgerufen; die Quelldateien befinden sich im Verzeichnis `/opt/lampp/htdocs/orchid`.

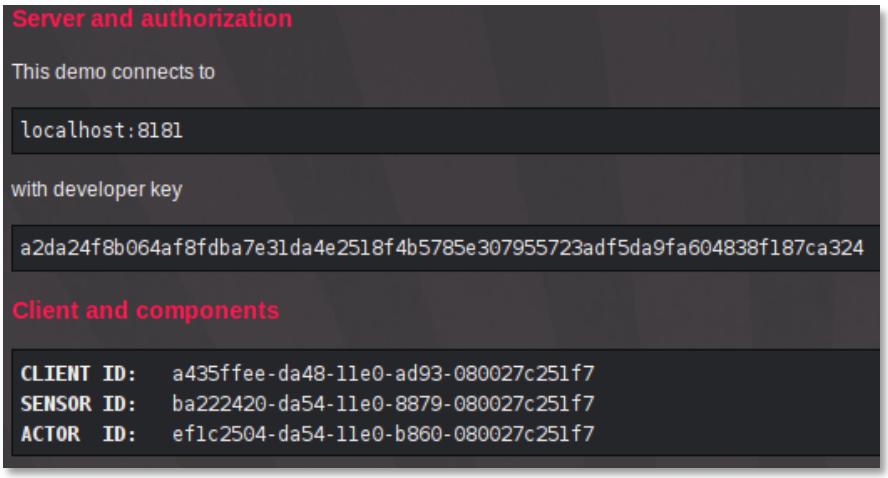

Abbildung 64 - Informationsbereich der Proof of Concept Drittsoftware

Die obige Abbildung zeigt einen Ausschnitt aus der Drittsoftware. Die dargestellten Parameter müssen den in Server und Client verwendeten Parametern entsprechen, damit die Software die Möglichkeiten des ORCHID Toolkits demonstrieren kann, und können bei Bedarf in der Datei `/opt/lampp/htdocs/orchid/demo/demo.php` angepasst werden.

Im unteren Teil der Anwendung befinden sich drei Bereiche, die im Folgenden erläutert werden.

10.8.1 Sensordaten abrufen

Der erste Bereich dient dem Abrufen von Sensordaten von einem ORCHID Server (in der Voreinstellung: `localhost`) und verfügt über die zwei in der folgenden Abbildung gezeigten Schaltflächen.

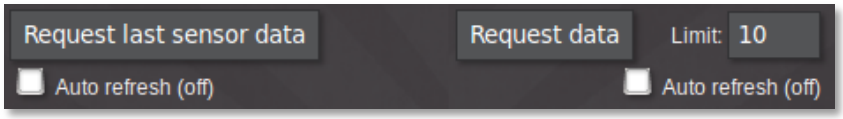

Abbildung 65 - Steuerelemente zum Abrufen von Sensordaten

Über den Button *„Request last sensor data“* wird der letzte Sensorwert, den ein Client für einen Sensor an den Server übermittelt hat, abgerufen. *„Request data“* ruft mehrere Sensorwerte ab (die Anzahl kann im nebenstehenden Eingabefeld bestimmt werden[77]).

Wird der letzte Sensorwert abgerufen, verändert sich die Anzeige wie in Abbildung 66 dargestellt.

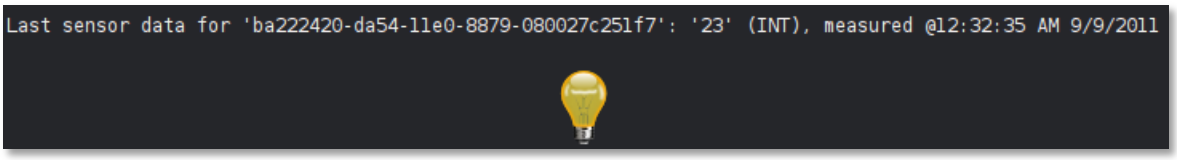

Abbildung 66 - Ausgabe eines abgerufenen Sensorwerts (1)

Der Ausgabe kann entnommen werden, dass der letzte für den Sensor mit der GUID `ba222420-da54-11e0-8879-080027c251f7` ermittelte Sensorwert `23` betrug und um am 09.09.2011 um 00:32:35 Uhr gemessen wurde. Die Drittsoftware interpretiert diesen Wert selbständig als „die Lampe ist eingeschaltet“ und zeigt entsprechend das in der obigen Abbildung dargestellte Bild an.

Ein höherer Sensorwert (der Schwellenwert kann in der Datei `orchid.php` angepasst werden) würde bedeuten, dass weniger Licht auf den Sensor fällt. Entsprechend würde sich auch die Darstellung verändern, wie in der folgenden Abbildung demonstriert wird.

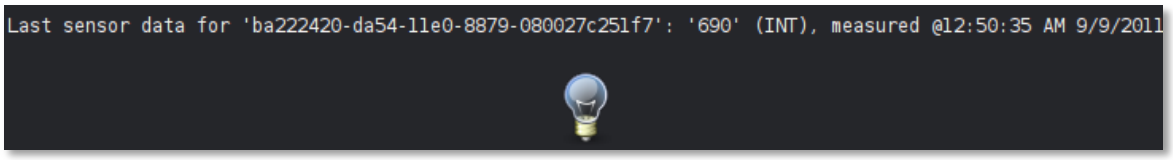

Abbildung 67 - Ausgabe eines abgerufenen Sensorwerts (2)

Werden mehrere Datensätze abgerufen, wird ein einfaches Diagramm angezeigt, in dem die gemessenen Werte auf der y-Achse, die Zeitpunkte der Messungen auf der x-Achse abgetragen sind. Je weiter rechts sich ein Datenpunkt in diesem Diagramm befindet, desto aktueller ist er.

Die folgende Abbildung 68 zeigt die Darstellung von zehn aufeinander folgenden Sensorwerten.

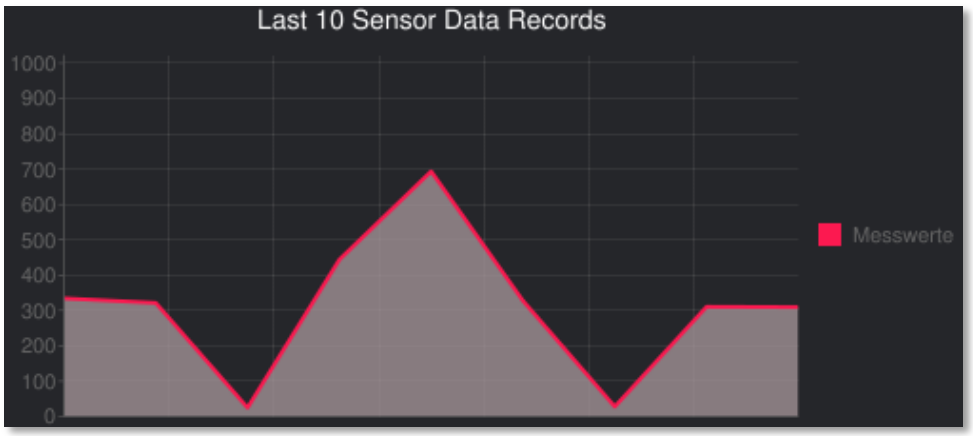

Abbildung 68 - Ausgabe mehrerer abgerufener Sensordaten in einem Diagramm

Unter jedem der beiden erläuterten Buttons befindet sich eine Schaltfläche *„Auto Refresh“*, mit der die automatische Aktualisierung der abgerufenen Sensordaten ein- und ausgeschaltet werden kann.

[77] Es können maximal 100 Sensorwerte abgerufen werden.

10.8.2 Instruktionen übermitteln

Der zweite Bereich, der sich auf der linken Seite im unteren Teil der Anwendung befindet, dient dazu, einem Client Instruktionen zu geben, indem diese an den ORCHID Server übermittelt werden.

Zu diesem Zweck enthält der Bereich drei Schaltflächen, die in der folgenden Abbildung gezeigt werden.

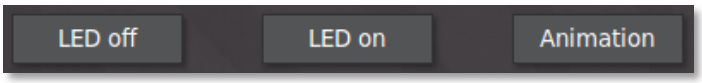

Abbildung 69 - Steuerelemente zum Senden von Instruktionen

Über den Button *„LED off“* kann eine Instruktion an den Server übermittelt werden, die dazu führt, dass die an den Client angeschlossene LED ausgeschaltet wird. *„LED on“* schaltet sie entsprechend ein. Die Schaltfläche *„Animation“* sendet sechs Instruktionen an den Server, die nacheinander vom Client angefordert und ausgeführt werden, was zu dreimaligem Ein- und Ausschalten der LED führt.

10.8.3 Client-Informationen abrufen

Der dritte und letzte Bereich befindet sich auf der rechten Seite im unteren Teil der Anwendung und ermöglicht das Abrufen von Informationen über einen ORCHID Client. Er enthält nur die Schaltfläche *„Get Arduino Info“*. Wird diese angeklickt, werden die dem Server bekannten Informationen über den angegebenen Client abgerufen und angezeigt, wie in der folgenden Abbildung 70 zu sehen ist.

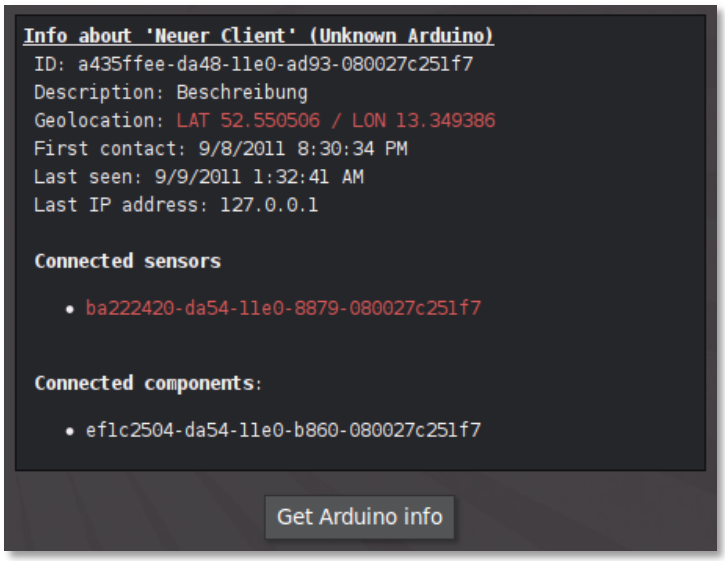

Abbildung 70 - Ausgabe von Client-Informationen

Verbundene Sensoren werden als Link dargestellt. Ein Klick auf diesen Link ruft den aktuellsten Sensorwert vom Server ab. Unter *„Geolocation“* wird die auf dem Server gespeicherte Position des Clients auf der Erde als Link auf Google Maps[78] angezeigt, über den die entsprechenden Koordinaten auf einer Weltkarte angezeigt werden können.

[78] http://maps.google.com/

10.9 Zusammenfassung

Zusammenfassend kann die Verwendung des ORCHID Toolkits mit den in der folgenden Tabelle genannten Schritten beschrieben werden, wobei die Installation entsprechender Software und ein vorhandener Developer Key vorausgesetzt werden.

#	Beschreibung
1	Einrichtung der MySQL-Datenbank auf dem Server und Import der Mindestdaten.
2	Erzeugung von GUIDs für jeden Client, Sensor und Aktor, der mit dem Toolkit verwendet werden soll.
3	Eintragung der GUIDs in die entsprechenden Datenbanktabellen auf dem Server.
4	Aufbau einer Schaltung mit den Sensoren und Aktoren, die dem Server bekannt sind.
5	Implementierung eines Sketchs, der die Sensoren und Aktoren anspricht und die Client-Bibliothek einbindet.
6	Festlegung der Berechtigungen für den Client auf dem Server.
7	Implementierung einer Drittsoftware, die mit einem ORCHID Server kommunizieren kann.

Tabelle 106 - Zusammenfassung der Toolkit-Verwendung

11 Zusammenfassung

Im Folgenden wird abschließend zusammengefasst, welche Schlüsse aus den im Verlauf dieser Arbeit gewonnen Erkenntnissen gezogen werden können, rückblickend die Auswahl der verwendeten Technologien bewertet und eine Aussage über den Erfolg der Arbeit getroffen.

Im Anschluss daran wird ein Ausblick auf das Weiterentwicklungspotential der im Rahmen der Arbeit entwickelten Software gegeben.

11.1 Rückblick

Im Verlauf dieser Arbeit wurde gezeigt, wie durch die Kombination verschiedener Technologien eine Werkzeugsammlung entwickelt werden kann, die die Integration nahezu beliebiger Hardware in nahezu beliebige Softwareprojekte und damit die Verbindung der realen Welt mit der Welt der Software ermöglicht.

Zunächst wurden dazu die verwendeten Technologien und die notwendigen technologischen Grundlagen beschrieben. Darauf aufbauend wurde im weiteren Verlauf der Arbeit ein Softwaresystem entworfen und implementiert, dass einerseits die geforderten Möglichkeiten bietet, andererseits auch für zukünftige Veränderungen und Erweiterungen gerüstet ist. Im Rahmen des Entwurfs und der Implementierung wurden weitere Technologien und Techniken erläutert und entwickelt, mit denen die Zukunftssicherheit und Flexibilität der entwickelten Werkzeuge sichergestellt werden konnte.

Die Wahl, Mono mit C# als Entwicklungsplattform für die Server-Komponente einzusetzen, kann rückblickend als richtige Entscheidung angesehen werden. C# bietet zahlreiche einzigartige Sprachkonstrukte, die in anderen Programmiersprachen und auf anderen Entwicklungsplattformen nicht zur Verfügung stehen, und ist diesen gegenüber entsprechend im Vorteil. Zusätzlich können die entstandenen Werkzeuge ohne jede Anpassung auf allen gängigen Betriebssystemen eingesetzt werden. Darüber hinaus existieren für die Plattform Mono einzigartige Werkzeuge, die für andere Plattformen wie Java nicht verfügbar sind. Diese Werkzeuge, von denen mehrere für die Entwicklung im Zuge dieser Arbeit eingesetzt wurden, zeichnen sich durch ihre Einfachheit aus, sind aber gleichzeitig robust und flexibel, so dass sie auch in einer professionellen Umgebung problemlos eingesetzt werden können.

Durch den Verzicht von Java als Programmiersprache und Entwicklungsplattform sind Entwickler, die das entstandene Toolkit erweitern oder verändern wollen, nicht an eine einzige Programmiersprache gebunden, sondern können dies in allen Sprachen tun, die von Mono unterstützt werden[79], (darunter auch Java[80]). Diese Eigenschaft der Mono-Plattform bestätigt deren Auswahl als Entwicklungsplattform zusätzlich.

Die Wahl der Technologien, mit denen die Client-Komponente entwickelt werden sollte, konnte nicht frei getroffen werden, da eine Bibliothek für einen Arduino Microcontroller unter Verwendung von C/C++ entwickelt werden muss. Daraus ergaben sich einige Probleme während der Entwicklung, die im Verlauf der Arbeit ebenfalls erläutert wurden.

Maßgeblich für die Vielfalt der Einsatzmöglichkeiten, die die entstandene Werkzeugsammlung bietet, ist die Entscheidung, die Kommunikation zwischen Verbindungspartnern über HTTP und damit über das Netzwerk oder das Internet zu ermöglichen. Jede moderne, etablierte Programmiersprache verfügt über entsprechende Schnittstellen und kann in Verbindung mit dem ORCHID Toolkit eingesetzt werden, weshalb diese Wahl als richtig anzusehen ist.

---

[79] http://www.mono-project.com/Languages
[80] http://www.mono-project.com/Java

Gravierende Probleme, die ein Scheitern zur Folge gehabt hätten, traten nicht auf. Im Hinblick auf das Ziel dieser Arbeit, nämlich die Entwicklung und Implementierung einer Werkzeugsammlung, über die Entwickler in die Lage versetzt werden, in ihren Softwareprojekten mit Hardware zu interagieren, wobei diese einerseits möglichst ohne großen Aufwand einsetzbar, andererseits aber sehr flexibel und erweiterbar sein sollte, ist das Ergebnis als Erfolg einzuschätzen.

11.2 Ausblick

Aufgrund der Auslegung des ORCHID Toolkits auf eine spätere Veränderung und dessen Einsatz in Drittsoftware-Projekten, ergibt sich ein großes Weiterentwicklungspotential, das durch die konsequente Anwendung objektorientierter Techniken und offener Frameworks zusätzlich verstärkt wird.

Einige der meiner Meinung nach interessantesten Möglichkeiten für potentielle Weiterentwicklungen des Toolkits werden im Folgenden beschrieben.

11.2.1 Nachrichten

Das Versenden von Nachrichten zwischen zwei ORCHID Clients (über einen ORCHID Server) ist ein optionales Wunschkriterium. Clients könnten mit dieser Erweiterung des Toolkits Daten austauschen, angefangen von einfachen Statusmeldungen, bis hin zum Austausch komplexer Datenstrukturen wie Sensordaten oder sogar Instruktionen. Ein Client wäre damit in der Lage, auch andere Clients zu instruieren, wenn ein Entwickler dies wünscht.

Die grundlegende Infrastruktur für die Nachrichtenübermittlung ist bereits Teil des Toolkits. Zur Umsetzung muss lediglich die entsprechende Programmlogik in die Server-Komponente und die Client-Bibliothek implementiert werden.

11.2.2 Drittsoftware

Ein langfristiges Ziel des ORCHID Toolkits, das den Rahmen dieser Arbeit jedoch gesprengt hätte, ist die Implementierung von Bibliotheken in möglichst vielen Programmiersprachen und für möglichst viele Plattformen, die es einer Drittsoftware erlauben, das Toolkit einzusetzen, um Hardware über das Netzwerk oder Internet zu integrieren und zu kontrollieren.

11.2.3 Smartphone-Applikationen

Moderne Smartphone-Betriebssysteme wie *Android*[81] (Google) oder *iOS*[82] (Apple) erlauben die Programmierung von Software für mobile Geräte (*„Apps“*). Apps sind oftmals in der Lage, Daten aus dem Internet zu beziehen oder Daten auf einen Server im Internet zu laden.

Entsprechend wäre eine Bibliothek, mit der eine Smartphone-App in die Lage versetzt würde, mit einem ORCHID Server zu kommunizieren, ein mächtiges Werkzeug, dessen Verfügbarkeit zur Entwicklung zahlreicher nützlicher Apps zur entfernten Steuerung und Überwachung von Hardware mit dem ORCHID Toolkit führen könnte.

11.2.4 Datenbank-Systeme

Durch die abstrakte Basisklasse, die dem verwendeten MySQL-Datenbankadapter zugrunde liegt, können weitere Datenbanksysteme (auch andere Speichersysteme sind denkbar) schnell und leicht implementiert werden.

Eine Erweiterung des Toolkits auf jedes beliebige (mit dem Mono Framework verwendbare) Datenbanksystem ist also möglich und ebenfalls ein langfristiges Ziel der Entwicklung des Toolkits.

[81] http://www.android.com/
[82] http://www.apple.com/ios/

Durch die Möglichkeit, auch andere Datenspeichersysteme verwenden zu können (z.B. Dateien), wird es auch Entwicklern, die nicht über den Zugriff auf eine Datenbank verfügen, ermöglicht, Projekte auf Basis des ORCHID Toolkits zu entwickeln.

12 Anhang I - Literaturverzeichnis

[1] Wikipedia. (2011, Juli) Wikipedia - Die freie Enzyklopädie. [Online].
http://de.wikipedia.org/wiki/Akronym

[2] Wikipedia. (2011, Juli) Wikipedia - Die freie Enzyklopädie. [Online].
http://de.wikipedia.org/wiki/Backronym

[3] Wikipedia. (2011, Juli) Wikipedia - Die freie Enzyklopädie. [Online].
http://de.wikipedia.org/wiki/Homophon

[4] Arduino. (2011, May) Arduino. [Online].
http://www.arduino.cc

[5] Arduino. (2011, Juli) Hardware. [Online].
http://arduino.cc/en/Main/Hardware

[6] Manuel Odendahl, Julian Finn, and Alex Wenger, *Arduino - Physical Computing für Bastler, Designer und Geeks*, 1. Auflage 2009. Köln, O'Reilly Verlag GmbH & Co. KG, 2009.

[7] Arduino. (2011, Januar) Arduino Board Duemilanove. [Online].
http://arduino.cc/en/Main/ArduinoBoardDuemilanove

[8] Atmel Corporation. (2011) Atmel AVR 8- and 32-bit - megaAVR. [Online].
http://www.atmel.com/dyn/products/devices.asp?category_id=163&family_id=607&subfamily_id=760

[9] Arduino. (2011, Mai) Arduino Ethernet Shield. [Online].
http://arduino.cc/en/Main/ArduinoEthernetShield

[10] Network Working Group. (1996, Mai) Hypertext Transfer Protocol -- HTTP/1.0. [Online].
http://www.w3.org/Protocols/rfc1945/rfc1945.txt

[11] The Internet Society. (1999) Hyptertext Transfer Protocol - HTTP/1.1. [Online].
http://www.w3.org/Protocols/rfc2616/rfc2616.html

[12] Novell, Inc. (2011) Languages - Mono. [Online].
http://mono-project.com/Languages

[13] Rose + Herleth GbR. EZcontrol Heimautomatisierung - Produkte. [Online].
http://www.ezcontrol.de/content/blogsection/4/28/

[14] digitalSTROM.org. (2010) digitalSTROM / Bilder. [Online].
http://www.digitalstrom.org/support/images.html

[15] Pachube.com. (2011) Press Room - Pachube. [Online].
https://pachube.com/press

[16] Stefan Matyba, *Entwicklung eines Frameworks zur Analyse, Synthese und Visualisierung von Graphen*. Berlin, Berlin, Technische Fachhochschule Berlin, 2008.

[17] Tobias Klein, *Buffer-Overflows und Format-String-Schwachstellen*, 1. Auflage 2004. Heidelberg, Deutschland, dpunkt.verlag GmbH, 2004.

[18] The Computer Language Benchmarks Game. (2011, August) C# Mono speed % Java 7 -server speed. [Online]. http://shootout.alioth.debian.org/u64q/benchmark.php?test=all&lang=csharp

[19] Creative Commons. Creative Commons Legal Code. [Online]. http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode

[20] Creative Commons. Creative Commons - Attribution-NonCommercial-ShareAlike 3.0 Unported - CC BY-NC-SA 3.0. [Online]. http://creativecommons.org/licenses/by-nc-sa/3.0/

[21] Arduino. (2009, Februar) Arduino - Ethernet. [Online]. http://www.arduino.cc/en/Reference/Ethernet

[22] Jon Erickson, *Hacking - Die Kunst des Exploits*. Heidelberg, Deutschland, dpunkt.verlag GmbH, 2009.

[23] Judith Bishop, *C# 3.0 Design Patterns*, 1. Edition., O'Reilly Media Inc., Ausgabe. USA, 2008.

[24] Erich Gamma, Richard Helm, Ralf Johnson, and John Vlissides, *Entwurfsmuster*, 5., korrigierter Nachdruck., Addison-Wesley Publishing Company, Ausgabe. Deutschland, 2001.

[25] The Internet Society. (1998, April) RFC 2324 - Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0). [Online]. http://tools.ietf.org/html/rfc2324

[26] Klaus Schmeh, *Kryptografie*, 3., überarbeitete und erweiterte Auflage 2007. Heidelberg, Deutschland, dpunkt.verlag GmbH, 2007.

[27] Jonathan Oxer. (2010) Arduino Shield List. [Online]. http://www.shieldlist.org/

[28] Adafruit Industries. Adafruit GPS logger shield kit [v1.1]. [Online]. http://www.adafruit.com/products/98

[29] Microsoft Corporation. Design Guidelines for Class Library Developers. [Online]. http://msdn.microsoft.com/de-de/library/czefa0ke(v=vs.71).aspx

[30] Arduino. (2010, September) Arduino - Getting Started. [Online]. http://arduino.cc/en/Guide/HomePage

[31] Microsoft Corporation. Threading (C#). [Online]. http://msdn.microsoft.com/en-US/library/ms173178(v=VS.80).aspx

13 Anhang II – Abbildungsverzeichnis

Abbildung 1 - Arduino "Duemilanove" 6
Abbildung 2 - Digitale I/O-Pins des Arduino Duemilanove 8
Abbildung 3 - ISO/OSI-Referenzmodell 9
Abbildung 4 - Screenshot "Fiddler" 11
Abbildung 5 - EZ Control Steuergerät "XS1" 17
Abbildung 6 - digitalSTROM-Chip 18
Abbildung 7 - pachube-Logo 19
Abbildung 8 - Logo der CC BY-NC-SA 3.0 26
Abbildung 9 - Blinkenlights 28
Abbildung 10 - Projektstruktur der Server-Komponente 50
Abbildung 11 - Namespaces in "Orchid.Core" 51
Abbildung 12 - Projektstruktur der Client-Bibliothek 52
Abbildung 13 - Hierarchie in der Server-Komponente 53
Abbildung 14 - Entwurfsmuster "Abstrakte Fabrik" 54
Abbildung 15 - Fabriken für Server und Daemons 55
Abbildung 16 - Vererbungshierarchie von Servern und Daemons 77
Abbildung 17 - Beziehungen der Daemon-Klasse 78
Abbildung 18 - Vererbungshierarchie von Parsern und Requests 78
Abbildung 19 - Beziehungen und Hierarchien der Datenhaltungsklassen 79
Abbildung 20 - Beziehungen der Datenbankadapter-Klasse 80
Abbildung 21 - Abhängigkeiten des Typeloading-Systems 80
Abbildung 22 - Klassen der Client-Bibliothek 81
Abbildung 23 - Lebenszyklus eines Threads 91
Abbildung 24 - Multithreading (OrchidHardwareServer) 93
Abbildung 25 - Allgemeine Bearbeitung von Requests 94
Abbildung 26 - Bearbeitung von Client-GET-Requests 95
Abbildung 27 - Bearbeitung von Client-PUT-Requests 96
Abbildung 28 - Bearbeitung von Client-TRACE-Requests 97
Abbildung 29 - Bearbeitung von Drittsoftware-GET-Requests 98
Abbildung 30 - Bearbeitung von Drittsoftware-PUT-Requests 99
Abbildung 31 - Bearbeitung von Drittsoftware-SEARCH-Requests 100
Abbildung 32 - Datenbankstruktur 101
Abbildung 33 - Typeloading 110
Abbildung 34 - Ausgabe von dkgen 113
Abbildung 35 - Bildschirmfoto "Wireshark" 117
Abbildung 36 - Installation von XAMPP for Linux 120
Abbildung 37 - Erzeugung der MySQL-Datenbank 121
Abbildung 38 - Import der Datenbankstruktur 121
Abbildung 39 - Referenzierung der Server-Komponente 121
Abbildung 40 - Referenzierungsdialog von MonoDevelop 122
Abbildung 41 - Referenzierte Server-Komponente 122
Abbildung 42 - Hauptfenster der Arduino IDE 137
Abbildung 43 - Bibliotheksimport in der Arduino IDE 138
Abbildung 44 - Ausgabe von ggen 151
Abbildung 45 - Erzeugung mehrerer GUIDs mit ggen 151
Abbildung 46 - Erzeugung eines Developer Keys mit dkgen 151
Abbildung 47 - Auswahl einer virtuellen Festplatte 152
Abbildung 48 - Desktop der virtuellen Maschine 153
Abbildung 49 - Starten von XAMPP (Linux) 154
Abbildung 50 - Anmeldung am lokalen Webserver 154
Abbildung 51 - Login in phpMyAdmin 155
Abbildung 52 - Dialog zur Erzeugung einer neuen Datenbank 155

Abbildung 53 - Dialog zum Import einer SQL-Datei 155
Abbildung 54 - Ausgabe von uuid 155
Abbildung 55 - Auflistung von Datenbanktabellen in phpMyAdmin 156
Abbildung 56 - Eingabemaske für neue Datensätze 156
Abbildung 57 - Hinzufügen eines neuen ORCHID Clients (Shellscript) 157
Abbildung 58 - Ausgabe des Referenzservers 157
Abbildung 59 - Schaltplan der Beispielschaltung 158
Abbildung 60 - Verweis auf einen Client 159
Abbildung 61 - Fehlercodes im seriellen Monitor der Arduino IDE 161
Abbildung 62 - Einfügen von Credentials in phpMyAdmin 161
Abbildung 63 - Ausgabe im seriellen Monitor der Arduino IDE 161
Abbildung 64 - Informationsbereich der Proof of Concept Drittsoftware 162
Abbildung 65 - Steuerelemente zum Abrufen von Sensordaten 162
Abbildung 66 - Ausgabe eines abgerufenen Sensorwerts (1) 163
Abbildung 67 - Ausgabe eines abgerufenen Sensorwerts (2) 163
Abbildung 68 - Ausgabe mehrerer abgerufener Sensordaten in einem Diagramm 163
Abbildung 69 - Steuerelemente zum Senden von Instruktionen 164
Abbildung 70 - Ausgabe von Client-Informationen 164

14 Anhang III – Tabellenverzeichnis

Tabelle 1 - Preisübersicht ATmega Microcontroller 5
Tabelle 2 - HTTP-Methoden 11
Tabelle 3 - Ausgewählte Header-Felder 12
Tabelle 4 - HTTP Status-Codes 13
Tabelle 5 - FS10: Serverkonfiguration einlesen 31
Tabelle 6 - FS20: Hardware-Server starten 31
Tabelle 7 - FS30: Software-Server starten 31
Tabelle 8 - FS40: Auf eingehende Client-Verbindung warten 31
Tabelle 9 - FS50: Auf eingehende Drittsoftware-Verbindung warten 32
Tabelle 10 - FS60: Eingehende Client-Verbindung annehmen 32
Tabelle 11 - FS70: Eingehende Drittsoftware-Verbindung annehmen 32
Tabelle 12 - FS80: Eingehende Anfrage auslesen 33
Tabelle 13 - FS90: Gelesene Anfrage transformieren 33
Tabelle 14 - FS100: Request-Typ bestimmen 33
Tabelle 15 - FS110: Sensordaten speichern 34
Tabelle 16 - FS120: Sensordaten abrufen 34
Tabelle 17 - FS130: Heartbeat-Request verarbeiten 34
Tabelle 18 - FS140: Client-Instruktion abrufen 35
Tabelle 19 - FS150: Client- Instruktion speichern 35
Tabelle 20 - FS160: Client-Informationen abrufen 35
Tabelle 21 - FS170: Ungültige oder fehlerhafte Anfrage behandeln 36
Tabelle 22 - FS180: Nachricht speichern (Drittsoftware) 36
Tabelle 23 - FS190: Nachricht speichern (Client) 36
Tabelle 24 - FS200: Nachricht an ORCHID Client übermitteln 37
Tabelle 25 - FS210: Berechtigungen für ORCHID Client abrufen 37
Tabelle 26 - FS220: Client-Instruktion speichern (Client) 38
Tabelle 27 - /FS230/ - Developer Key prüfen 38
Tabelle 28 - FC10: ORCHID Client-Instanz erzeugen 39
Tabelle 29 - FC20: Sensorwert an ORCHID Server übermitteln 39
Tabelle 30 - FC20a: Integer übermitteln 39
Tabelle 31 – FC20b: Float übermitteln 39
Tabelle 32 - FC20c: Double übermitteln 39
Tabelle 33 - FC20d: Unsigned Integer übermitteln 39
Tabelle 34 - FC20e: Character übermitteln 40
Tabelle 35 - FC20f: Boolean übermitteln 40
Tabelle 36 - FC20g: Zeichenkette übermitteln 40
Tabelle 37 - FC20h: Byte übermitteln 40
Tabelle 38 - FC20i: Long übermitteln 40
Tabelle 39 - FC20j: Unsigned Long übermitteln 40
Tabelle 40 - FC30: Heartbeat an ORCHID Server senden 40
Tabelle 41 - FC40: Instruktion für Aktor abrufen 41
Tabelle 42 - FC50: Instruktion für Aktor ausführen 41
Tabelle 43 - FC60: Anfrage an ORCHID Server senden 41
Tabelle 44 - FC70: Auf eingehende Server-Antwort warten 42
Tabelle 45 - FC80: Eingehende Server-Antwort auslesen 42
Tabelle 46 - FC90: Nachricht auf ORCHID Server hochladen 42
Tabelle 47 - FC100: Nachricht von ORCHID Server abrufen 43
Tabelle 48 - FC110: Instruktion an ORCHID Server senden 43
Tabelle 49 - DS10: Aktoren 43
Tabelle 50 - DS20: Sensoren 43
Tabelle 51 – DS30: Arduinos 44
Tabelle 52 – DS40: Berechtigungen 44
Tabelle 53 – DS50: Geolocations 44

TABELLE 54 - DS60: SENSORDATEN 44
TABELLE 55 – DS70: INSTRUKTIONEN 45
TABELLE 56 - DS80: NACHRICHTEN 45
TABELLE 57 - DC10: INSTRUKTIONEN 45
TABELLE 58 - LS10: ANTWORTZEITEN 46
TABELLE 59 - LS20: BETRIEBSDAUER 46
TABELLE 60 - LS30: BELASTBARKEIT 46
TABELLE 61 - LS40: DATENGENAUIGKEIT 47
TABELLE 62 - LS50: ROBUSTHEIT 47
TABELLE 63 - LC10: BETRIEBSDAUER 47
TABELLE 64 - LC20: DATENGENAUIGKEIT 47
TABELLE 65 - LC30: ROBUSTHEIT 48
TABELLE 66 - NICHT VERWENDETE FXCOP-REGELN 75
TABELLE 67 - NICHT VERWENDETE STYLECOP-REGELN 76
TABELLE 68 - PROTOKOLL: SENSORDATEN HOCHLADEN 83
TABELLE 69 - CODES FÜR DATENTYPEN 84
TABELLE 70 - PROTOKOLL: HEARTBEAT SENDEN 84
TABELLE 71 - PROTOKOLL: INSTRUKTION ABRUFEN 85
TABELLE 72 - CODES FÜR AUSZUFÜHRENDE AKTIONEN 86
TABELLE 73 - CODES FÜR DATENTYPEN 86
TABELLE 74 - PROTOKOLL: INSTRUKTIONEN HOCHLADEN 87
TABELLE 75 - PROTOKOLL: SENSORDATEN ABRUFEN 88
TABELLE 76 - PROTOKOLL: CLIENT-INFORMATIONEN ABRUFEN 89
TABELLE 77 - DATENBANKTABELLE "ACTIONS" 102
TABELLE 78 - DATENBANKTABELLE "ACTORS" 102
TABELLE 79 - DATENBANKTABELLE "ARDUINOS" 102
TABELLE 80 - DATENBANKTABELLE "CREDENTIALS" 103
TABELLE 81 - DATENBANKTABELLE "INSTRUCTIONS" 103
TABELLE 82 - DATENBANKTABELLE "KNOWN_HOSTS" 104
TABELLE 83 - DATENBANKTABELLE "MESSAGES" 104
TABELLE 84 - DATENBANKTABELLE "SENSORS" 104
TABELLE 85 - DATENBANKTABELLE "SENSOR_DATA" 105
TABELLE 86 - DATENBANKTABELLE "TYPEMAPPINGS" 105
TABELLE 87 - NAMENSKONVENTIONEN 107
TABELLE 88 - BINÄRDATEIEN DER SERVER-KOMPONENTE 119
TABELLE 89 – EINSTELLUNGSKATEGORIEN 124
TABELLE 90 - EINSTELLUNGSKATEGORIE "DEPENDENCIES" 127
TABELLE 91 - EINSTELLUNGSKATEGORIE "SERVER" 127
TABELLE 92 - EINSTELLUNGSKATEGORIE "RESTRICTIONS" 127
TABELLE 93 - EINSTELLUNGSKATEGORIE "LOGGING" 128
TABELLE 94 - EINSTELLUNGSKATEGORIE "DATABASE" 128
TABELLE 95 - EINSTELLUNGSKATEGORIE "LOCALE" 129
TABELLE 96 - EINSTELLUNGSKATEGORIE "VERBOSITY" 129
TABELLE 97 - EINSTELLUNGSKATEGORIE "DEBUG" 129
TABELLE 98 - QUELLDATEIEN DER CLIENT-BIBLIOTHEK 139
TABELLE 99 - KONSTRUKTORPARAMETER DER KLASSE ORCHID 140
TABELLE 100 - ACTION-CODES UND ENTSPRECHENDE KONSTANTEN 145
TABELLE 101 - FEHLERCODES 147
TABELLE 102 - EINSTELLUNGEN DER VIRTUELLEN MASCHINE 152
TABELLE 103 - DESKTOP-ELEMENTE 153
TABELLE 104 - AUF DER VM INSTALLIERTE SOFTWARE 154
TABELLE 105 - BEISPIEL-GUIDS 159
TABELLE 106 - ZUSAMMENFASSUNG DER TOOLKIT-VERWENDUNG 165
TABELLE 107 - INHALT DER BEILIEGENDEN DVD I

15 Anhang IV – Listingverzeichnis

Listing 1 - Struktur einer HTTP-Nachricht 10
Listing 2 - Übersicht HTTP-Methoden 12
Listing 3 - Struktur eines HTTP-Headers 12
Listing 4 - Serverantwort "200 OK" 13
Listing 5 - Serverantwort "404 Not Found" 13
Listing 6 - Beispielmethode in mehreren Programmiersprachen 15
Listing 7 - Erzeugung eines Standard-ORCHID Servers 55
Listing 8 - Eine neue Daemon-Fabrik 56
Listing 9 - Veränderte Erzeugung eines ORCHID Servers 56
Listing 10 - Nicht threadsichere Singleton-Klasse 57
Listing 11 - Threadsichere Singleton-Klasse 58
Listing 12 - Beispielklasse ohne Properties 61
Listing 13 - Anwendung von Zugriffsmethoden 61
Listing 14 - Beispielklasse mit Properties 62
Listing 15 - Anwendung von Properties 62
Listing 16 - Beispielklasse mit unterschiedlichen Zugriffsmodifizierern 62
Listing 17 - Beispielhafte Extension-Method 63
Listing 18 - Anwendung einer Extension Method 63
Listing 19 - Extension Method mit Lambda Expression 64
Listing 20 - Exemplarische Lambda Expression 64
Listing 21 - MergeAndFilter mit Schleifen 65
Listing 22 - MergeAndFilter mit LINQ 65
Listing 23 - FilterAndSort mit LINQ 66
Listing 24 - Beispielhafte NLog-Konfigurationsdatei 68
Listing 25 - NLog-Konfiguration aus dem Code 69
Listing 26 - Beispielhafte Log-Meldung 70
Listing 27 - Beispiel für ein NinjectModule 71
Listing 28 - Erzeugung eines Ninject-Kernels 72
Listing 29 - Objekterzeugung mit Ninject 72
Listing 30 - Laden einer Konfigurationsdatei mit Nini 73
Listing 31 - Beispiel für eine INI-Datei 73
Listing 32 - Verwendung von Nini 74
Listing 33 - Allgemeine Antwort eines ORCHID Servers 82
Listing 34 - Übermittlung einer Instruktion (Server-Antwort) 85
Listing 35 - Übermittlung von Sensordaten (Server-Antwort) 89
Listing 36 - Übermittlung von Client-Informationen (Server-Antwort) 90
Listing 37 - Übermittlung eines einzelnen Sensorwerts (Server-Antwort) 98
Listing 38 - Übermittlung mehrerer Sensorwerte (Server-Antwort) 98
Listing 39 - Übermittlung von Client-Informationen (Server-Antwort) 101
Listing 40 - Styleguide zu Einrückungen und Klammern 106
Listing 41 - Installationsbefehl für uuid (apt-get) 108
Listing 42 - Beispiel für einfaches Typeloading 109
Listing 43 - Minimalprojekt (ORCHID Server) 123
Listing 44 - Initialisierung des Logging-Systems 130
Listing 45 - Verbessertes Minimalprojekt (ORCHID Server) 131
Listing 46 - Grundgerüst "CustomHardwareDaemon" 132
Listing 47 - Konstruktor "CustomHardwareDaemon" 133
Listing 48 - Überschreiben von "HandleRequest" 133
Listing 49 - Methode zur Behandlung von TRACE-Requests 134
Listing 50 - Anpassung der Konfigurationsdatei 134
Listing 51 - Veränderung der Methode zur Anfragebehandlung 135
Listing 52 – Ungültige Methode zur Behandlung von HEAD-Requests 135

Listing 53 - Gültige Methode zur Behandlung von HEAD-Requests 136
Listing 54 - Pfad zum Sketch-Verzeichnis unter Microsoft Windows 138
Listing 55 - Pfad zum Sketch-Verzeichnis unter Linux 138
Listing 56 - Referenzierung der Client-Bibliothek 139
Listing 57 - Grundgerüst eines Sketchs 140
Listing 58 - Einbindung der Client-Bibliothek 140
Listing 59 - Erzeugung eines Orchid-Objekts (1) 140
Listing 60 - Erzeugung eines Orchid-Objekts (2) 141
Listing 61 - Initialisierung eines Orchid-Objekts 142
Listing 62 - Senden eines Heartbeats 142
Listing 63 - Hochladen von Sensordaten 143
Listing 64 - Initialisierung zum Abrufen von Instruktionen 144
Listing 65 - Abrufen einer Instruktion 145
Listing 66 - Direktes Ausführen einer Instruktion 146
Listing 67 - Fehlermeldung der Arduino IDE 147
Listing 68 - Fehlerbehebung durch Einbindung von Ethernet.h 147
Listing 69 - Beispielhaftes Header-Feld zur Inhaltsverschlüsselung 148
Listing 70 - Hinzufügen des neuen Header-Felds (1) 148
Listing 71 - Hinzufügen des neuen Header-Felds (2) 149
Listing 72 - Erweiterung der Client-Komponente 150
Listing 73 - Beispielsketch 160

16 Anhang V – Glossar

Aktor	Im Gegenteil zu einem → *Sensor* ist ein *Aktor* eine → *Komponente*, die mit der Umgebung interagieren und diese manipulieren kann, z.B. ein Motor oder eine → *LED*.
Anti-Pattern	Das Gegenteil zu einem Design Pattern (Entwurfsmuster). Beschreibt eine sich in der Programmierung oftmals wiederholende schlechte Lösung für ein bestimmtes Problem.
API	***A****pplication* ***P****rogramming* ***I****nterface*
Buffer Overflow	Ein *Buffer Overflow* tritt auf, wenn ein Wert in einen Speicher-Puffer fester Größe geschrieben wird und über dessen Grenzen hinaus läuft.
CamelCase → *PascalCase*	Eine in der Informatik verbreitete Schreibweise für aus mehreren Einzelwörtern zusammengesetzte Wörter. Die Trennung der einzelnen Wortbestandteile wird im *CamelCase* nicht wie üblich über ein Leerzeichen, sondern über Großbuchstaben innerhalb des Wortes erreicht.
CC-BY-NC-SA 3.0	Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported, eine Lizenz für Softwareprodukte. *http://creativecommons.org/licenses/by-nc-sa/3.0/*
CCC	***C****haos* ***C****omputer* ***C****lub* *http://www.ccc.de*
CIL	***C****ommon* ***I****ntermediate* ***L****anguage*; eine vom Compiler des Mono Frameworks aus dem Quellcode einer Anwendung erzeugte Zwischensprache, die vom → *JIT-Compiler* der → *CLR* bzw. der → *Mono Runtime* in ausführbaren Maschinencode übersetzt wird.
CLI	***C****ommon* ***L****anguage* ***I****nfrastructure*; beschreibt eine allgemeine Entwicklungsinfrastruktur für Anwendungen, die bezüglich ihrer Ausführung unabhängig von der verwendeten Programmiersprache und der ausführenden Systemumgebung sind.
Client	In einer Client-Server-Infrastruktur ist ein Client derjenige Kommunikationspartner, der eine Anfrage an einen → *Server* sendet, also derjenige, der die Kommunikation initialisiert.
CLR	***C****ommon* ***L****anguage* ***R****untime*. Siehe → *Mono Framework*.

Credentials	Als *Credentials* werden im Rahmen dieser Arbeit die Berechtigungen eines → *ORCHID Clients* auf einem → *ORCHID Server* bezeichnet.
CRLF	Die Zeichenfolge *„Carriage Return“*, gefolgt von *„Line Feed“* („`\r\n`“). Wird als Zeilenumbruch verwendet und trennt in HTTP-Transaktionen → *Header* voneinander, sowie den → *Message Header* vom → *Message Body*.
Daemon	In UNIX und ähnlichen Systemen wird ein Systemprozess, der im Hintergrund läuft, als *Daemon* bezeichnet. Im Rahmen dieser Arbeit werden Daemons von → *ORCHID Servern* erzeugt, um Anfragen von → *ORCHID Clients* zu verarbeiten und zu beantworten.
Delegate	Ein Objekt, das als *„Mittelsmann“* zwischen zwei anderen Objekten auftritt, wird als *Delegate* bezeichnet.
Dependency Injection	*Dependency Injection* ist eine auf → *Inversion of Control* basierende Technik zur Reduzierung starrer Abhängigkeiten innerhalb objektorientiertes Codes.
Developer Key	Eine Zeichenfolge, die zwischen → *ORCHID Client* und → *ORCHID Server* übertragen wird, um einen Client bei einem Server zu als zugriffsberechtigt zu autorisieren.
DI	Siehe → *Dependency Injection*.
DLL	***D****ynamic* ***L****ink* ***L****ibrary;* eine Sammlung von Funktionen und/oder Klassen in einer binären Datei.
Drittsoftware	Im Kontext dieser Arbeit wird jede Software, die das ORCHID Toolkit einsetzt, um mit Hardware zu kommunizieren, als *Drittsoftware* bezeichnet.
Framework	Ein *Framework* ist eine Sammlung von Bibliotheken und/oder Programmen.
Gateway	Als *Gateway* wird in einem Netzwerk ein Teilnehmer bezeichnet, der anderen Teilnehmern als zentraler Zugriffspunkt in ein anderes Netzwerk dient.
Geolocation	Die Koordinaten eines Ortes auf der Welt.

→ *GPS*

GPS	***G**lobal **P**ositioning **S**ystem*; ein planetenumspannendes System von Satelliten, die die Lokalisierung eines Ortes auf der Welt auf wenige Meter genau ermöglichen.
GUID	***G**lobally **U**nique **I**dentifier*; eine Implementierung des → *UUID*-Standards. Eine *GUID* ist eine 128-Bit-Zahl, die als eindeutige Kennung für Objekte genutzt werden kann.
Hardware-Daemon → *Software-Daemon*	Ein *Hardware-Daemon* ist ein → *ORCHID Daemon*, der mit → *ORCHID Clients* kommuniziert.
Hardware-Server → *Software-Server*	Ein *Hardware-Server* ist ein → *ORCHID Server*, der Anfragen von → *ORCHID Clients* entgegennimmt.
Header	Kurzform der Bezeichnung für Header-Felder innerhalb eines → *HTTP-Requests* oder einer → *HTTP-Response*.
Heartbeat	Ein von einem → *ORCHID Client* gesendeter TRACE-Request wird als Heartbeat bezeichnet und dient dazu, einem → *ORCHID Server* die Bereitschaft des Clients anzuzeigen.
HTML	***H**ypertext **M**arkup **L**anguage*; eine Auszeichnungssprache für Text. Wird hauptsächlich im Internet zur Auszeichnung von Webseiten verwendet.
HTTP	***H**ypertext **T**ransfer **P**rotocol*; ein in → *RFC* 2616 definierter Standard zur Kommunikation im Netzwerk.
HTTP-Code	Ein in einer → *HTTP-Response* angegebener *Code*, über den auf Erfolg oder Misserfolg einer Operation geschlossen werden kann.
HTTP-Methode	Die *Methode* eines → *HTTP-Requests* definiert die auszuführenden Aktionen, wenn ein solcher Request auf einem Server eingeht.
IDE	***I**ntegrated **D**evelopment **E**nvironment*; eine Software, die zur Entwicklung von Software genutzt wird und neben zahlreichen Werkzeugen auch den Compiler komfortabel integriert.
INI	Ein Dateiformat zur Speicherung von Schlüssel-Wert-Paaren, optional aufgeteilt in Kategorien.

Inversion of Control	Als *Inversion of Control* wird eine Technik der objektorientierten Programmierung beschrieben, bei der Methoden nicht wie üblich direkt aufgerufen, sondern bei einer zentralen Stelle registriert werden, die ihrerseits den Aufruf dieser Methoden übernimmt. Der ursprüngliche Aufrufer teilt der zentralen Stelle mit, dass eine Methode aufgerufen werden soll, ruft sie aber nicht mehr selbst auf.
I/O-Board	Als *I/O-Board* wird im Kontext dieser Arbeit ein Arduino Microcontroller-Board bezeichnet.
IoC	Siehe → *Inversion of Control*.
I/O-Pin	Ein → *I/O-Board* verfügt über digitale und analoge Ein- und Ausgänge, die als *I/O-Pins* bezeichnet werden.
IP-Spoofing	*IP-Spoofing* bezeichnet einen Prozess, bei dem ein Angreifer in einem Netzwerk Anfragen mit gefälschter Absender-IP-Adresse an einen Server sendet.
ISO	***I**nternational **O**rganization for **S**tandardization*. *http://www.iso.org*
ISO/OSI-Referenzmodell	***O**pen **S**ystems **I**nterconnection* Modell der → *ISO*; beschreibt Abstraktionsebenen in der Netzwerkkommunikation.
JIT-Compiler	Ein ***J**ust-**I**n-**T**ime-Compiler* übersetzt Quelltext erst unmittelbar vor dessen Ausführung in die Zielsprache.
Komponente	Eine *Komponente* ist ein → *Sensor* oder ein → *Aktor*.
Lambda Expression	Anonyme Funktionen, die als Parameter an eine Methode übergeben werden können werden in der → *CLI* als *Lambda Expression* bezeichnet.
Lambda Operator	Der *Lambda-Operator „=>“* kann innerhalb einer → *Lambda Expression* als Zuweisungsoperator verwendet werden.
LAMPP	*LAMPP* ist die für Linux-Systeme verfügbare Version von → *XAMPP*.
LC-Display	***L**iquid **C**rystal **D**isplay*; ein Display, das zur Anzeige von Daten Flüssigkristalle verwendet, die ihre Eigenschaften unter Spannung verändern.

LED	**L**ight **E**mitting **D**iode; eine → *Komponente* (→ *Aktor*), die bei anliegender Spannung Licht aussendet.
LINQ	***L**anguage **I**ntegrated **Q**uery*; ein Sprachkonstrukt der → *CLI*, das es erlaubt, → *SQL*-ähnliche Abfragen auf Objekte aus dem Quellcode einer Anwendung heraus durchzuführen.
MAC-Adresse	***M**edia-**A**ccess-**C**ontrol-Adresse*; eine eindeutige Kennung eines netzwerkfähigen Geräts.
Message Body	Der in einem → *HTTP-Request* oder einer → *HTTP-Response* übermittelte Inhalt wird als *Message Body* bezeichnet.
Message Header	Innerhalb eines → *HTTP-Requests* oder einer → *HTTP-Response* können Zusatzinformationen über die im → *Message Body* übermittelten Daten in → *Headern* übertragen werden, die als *Message Header* bezeichnet werden.
Mono Runtime	Die Laufzeitumgebung des Mono Frameworks. Stellt Klassenbibliotheken und eine Implementierung der → *CLR* zur Verfügung, um Anwendungen, die für die → *CLI* entwickelt wurden, auszuführen.
Mutex	***Mut**ual **Ex**clusion („gegenseitiger Ausschluss“)*; ein Mutex-Objekt wird eingesetzt, um den konkurrierenden Zugriff mehrerer Objekte auf einer Ressource zu verhindern.
ORCHID	Der Name des im Zuge dieser Arbeiten entwickelten Toolkits; ein Homophon auf → *ORCIT*.
ORCHID Client	Ein Arduino-Microcontroller, der die Client-Bibliothek des ORCHID Toolkits einsetzt, wird als *ORCHID Client* bezeichnet.
ORCHID Daemon	Ein ORCHID Daemon ist eine Instanz einer Klasse, die zur Verarbeitung eingehender → *Requests* von → *ORCHID Clients* oder → *Drittsoftware* eingesetzt wird.
ORCHID Server	Eine Instanz der Server-Komponente des ORCHID Toolkits wird als *ORCHID Server* bezeichnet.
ORCIT → *ORCHID*	***O**nline **R**eal-**L**ife **C**lients **I**nternet-**T**oolkit*; ein Akronym, das namensgebend für das ORCHID Toolkit ist.

PascalCase *→ CamelCase*	*PascalCase* bezeichnet eines Sonderform des → *CamelCase*, bei der der erste Buchstabe eines Worts stets ein Großbuchstabe sein muss.
PHP	***H****ypertext* ***Prep****rocessor*; eine Programmiersprache für das Internet/Netzwerk. *http://www.php.net*
Puffer-Überlauf	Siehe → *Buffer Overflow*.
Request	Eine HTTP-Anfrage eines → Clients wird auch als *Request* bezeichnet.
Response	Die *Response* ist die Antwort eines → Servers auf einen → *HTTP-Request*.
RFC	***R****equest* ***F****or* ***C****omment*; ein Werkzeug zur Bewertung von vorgeschlagenen Internet-Standards.
RISC	***R****educed* ***I****nstruction* ***S****et* ***C****omputer*; eine Prozessorarchitektur, in der Befehle innerhalb eines Prozessortakts abgearbeitet werden können.
Sensor	Im Gegenteil zu einem → *Aktor* nimmt ein *Sensor* seine Umgebung wahr.
Server	In einer Client-Server-Infrastruktur ist ein *Server* derjenige Kommunikationspartner, der auf eingehende Anfragen eines → *Clients* wartet, diese bei Eingang entgegennimmt, verarbeitet und eine Antwort sendet.
Shield	Ein *Shield* ist eine aufsteckbare Erweiterungsplatine für einen Arduino Microcontroller.
Sketch	Programme für einen Arduino Microcontroller werden als *Sketches* bezeichnet.
Software-Daemon	Ein *Software-Daemon* ist ein → *ORCHID Daemon*, der mit → *Drittsoftware* kommuniziert.
Software-Server	Ein *Software-Server* ist ein → *ORCHID Server*, der Anfragen von → *Drittsoftware* entgegennimmt.
SQL	***S****tructured* ***Q****uery* ***L****anguage*; eine weit verbreitete Abfragesprachen für Datenbanksysteme.

Subnetzmaske	Die *Subnetzmaske* gibt für ein Netzwerk an, welche IP-Adressen für dieses Netz verfügbar sind.
TCP	***T**ransmission **C**ontrol **P**rotocol*; ein in → *RFC* 793 definiertes Protokoll für die paketbasierte Netzwerkkommunikation. *TCP* gilt als Basis des Internets.
TEA	***T**iny **E**ncryption **A**lgorithm*; ein sehr kompakter Verschlüsselungsalgorithmus, der nicht mehr als sicher gilt.
Toolkit	Als *Toolkit* wird eine Werkzeugsammlung bezeichnet, also eine Reihe von Programmen oder anderen Werkzeugen, die zusammen eine bestimmte Aufgabe erfüllen können.
TWAIN	***T**echnology **W**ithout **A**n **I**nteresting **N**ame*; ein Standard zum Datenaustausch zwischen bildgebenden Geräten (z.B. Scannern) und Computern.
URI	***U**niform **R**esource **I**dentifier*; eine Zeichenfolge, die eine bestimmte Ressource eindeutig identifiziert.
URL	***U**niform **R**esource **L**ocator*; eine Sonderform des → *URI*. In einem *URL* ist zusätzlich das zu verwendende Protokoll (z.B. → *HTTP*) angegeben.
User-Agent	Ein → *Client*, der über → *HTTP* auf eine Ressource zugreifen möchte, übermittelt (optional) eine Zeichenfolge, die dessen Identifikation darstellt; diese wird als *User-Agent* bezeichnet.
UUID	***U**niversally **U**nique **Id**entifier*; ein Standard zur Erzeugung eindeutiger Kennungen. Eine Umsetzung ist → *GUID*.
VM	***V**irtuelle **M**aschine*; als virtuelle Maschine wird eine Software bezeichnet, die in der Lage ist, ein Betriebssystem auszuführen, indem sie diesem vortäuscht, ein kompatibler Computer zu sein.
XAMPP	*XAMPP* ist ein Serverpaket, das unter anderem einen Apache2-Webserver, einen MySQL-Server und → *PHP* enthält. *http://www.apachefriends.org*
XML	*e**X**tensible **M**arkup **L**anguage*; eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten.
XTEA	*e**X**tended **T**iny **E**ncryption **A**lgorithm*; eine Weiterentwicklung von → *TEA*, die bisher als sicher gilt.

17 Anhang VI - Inhalt Datenträger

Der Datenträger, der dieser Arbeit beiliegt, enthält alle zur Verwendung des ORCHID Toolkits benötigten Dateien und Programme. Zusätzlich ist eine mit Oracle VirtualBox nutzbare virtuelle Maschine (vgl. Abschnitt 10) enthalten.

Die folgende Tabelle zeigt die auf der DVD enthaltenen Verzeichnisse und beschreibt, welche Dateien in ihnen enthalten sind.

Verzeichnis	Beschreibung
Arduino IDE	Enthält die aktuelle Version 0022 der Arduino IDE für Linux, Windows und MacOS.
Dokumentation	Enthält die automatisch aus den Quellcodekommentaren generierte Dokumentation.
Dokumente	Enthält die PDF-Version dieses Dokuments und verwendete Abbildungen in voller Auslösung.
Orchid	Enthält alle Dateien des ORCHID Toolkits
Client	Enthält die Client-Bibliothek.
examples	Enthält verschiedene Codebeispiele für die Verwendung der Client-Bibliothek.
lib	Enthält die eigentliche Client-Bibliothek zur Verwendung in einem Sketch.
Database	Enthält die Datenbankstruktur und benötigte Mindestdaten.
Server	Enthält die Server-Komponente.
bin	Enthält die kompilierte Version der Server-Komponente, die direkt in eigenen Projekten verwendet werden kann, sowie die Referenzimplementierung eines ORCHID Servers.
src	Enthält den vollständigen Quellcode der Server-Komponente.
Tests	Enthält Tests für Server-Komponente und Client-Bibliothek.
Tools	Enthält die Zusatztools dkgen und ggen.
src	Enthält den Quellcode von dkgen und ggen.
bin	Enthält die kompilierte und ausführbare Version von dkgen und ggen.
Quellen	Quellen, die im Verlauf der Arbeit genannt wurden, und Bildschirmfotos von genannten Internetseiten.
Runtime	Enthält die Mono Laufzeitumgebung für Linux (Installationsscript, benötigt eine aktive Internetverbindung und Administratorrechte) und Windows.
VM	Enthält die virtuelle Maschine.

Tabelle 107 - Inhalt der beiliegenden DVD